Cognex MVS-8000 Series

CVL Class Reference

CVL 8.0

June 2016

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Preface

This manual contains reference information for the Cognex Vision Library (CVL). CVL is a powerful C++ class library you can use to write machine vision applications for the Cognex MVS-8000 family of frame grabbers.

Style Conventions Used in This Manual

This manual uses the style conventions described in this section for text and software diagrams.

Text Style Conventions

This manual uses the following style conventions for text:

boldface	Used for C/C++ keywords, function names, class names, structures, enumerations, types, and macros. Also used for user interface elements such as button names, dialog box names, and menu choices.
italic	Used for names of variables, data members, arguments, enumerations, constants, program names, file names. Used for names of books, chapters, and sections. Occasionally used for emphasis.
`courier`	Used for C/C++ code examples and for examples of program output.
`bold courier`	Used in illustrations of command sessions to show the commands that you would type.
<italic>	When enclosed in angle brackets, used to indicate keyboard keys such as *<Tab>* or *<Enter>*.

Microsoft Windows Support

Cognex CVL software runs on Windows operating systems. In this documentation set, these are abbreviated to Windows unless there is a feature specific to one of the variants. Consult the *Getting Started* manual for your CVL release for details on the operating systems, hardware, and software supported by that release.

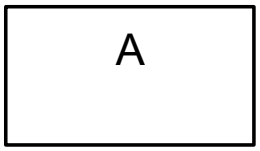
Software Diagramming Conventions

This manual uses the following symbols in class diagrams:

- **Classes** are shown as a box with the class name centered inside the box. For example, a class A with the C++ declaration

  ```
  class A{};
  ```

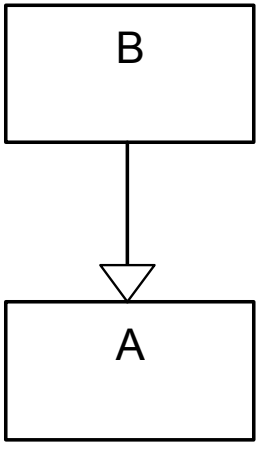
  is shown graphically as follows:



- **Inheritance** relationships between classes are shown using solid-line arrows from the derived class to the base class with a large, hollow triangle pointing toward the base class. For example, a class B that inherits from a class A with the declaration

  ```
  class B : public A {};
  ```

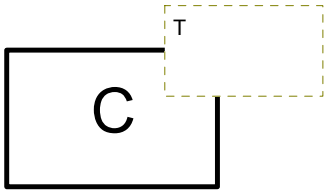
  is shown graphically as follows:



- **Template classes** are shown as a class box with a smaller, dotted-line rectangle representing the template parameter superimposed on the upper right corner of the class box. For example, a template class C with a parameter of type class T with the declaration:

  ```
  template <class T>
  class C{};
  ```

  is shown graphically as follows:



These symbols are based on the Unified Modeling Language (UML), a standard graphical notation for object-oriented analysis and design. See the latest *OMG Unified Modeling Language Specification* (available from the Object Management Group at http://www.omg.org) for more information.

Cognex Offices

Cognex Corporation serves its customers from the following locations:

Corporate Headquarters	Cognex Corporation Corporate Headquarters One Vision Drive Natick, MA 01760-2059 (508) 650-3000
Web Site	http://www.cognex.com

cc1Xform

```
#include <ch_cvl/xform.h>

class cc1Xform;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a one-dimensional transformation object You can use it to scale and offset points along one axis.

Constructors/Destructors

cc1Xform

```
cc1Xform();

cc1Xform(double scale, double offset);
```

- `cc1Xform();`

  Creates a new transformation object. Points are not scaled and not offset.

- `cc1Xform(double scale, double offset);`

  Creates a new transformation object in which points are scaled by *scale* and offset by *offset*.

  Parameters

scale	The amount by which to scale a point.
offset	The amount by which to offset the scaled point.

Operators

operator*

```
cc1Xform operator* (const cc1Xform& xform) const;

double operator* (double pt) const;
```

- `cc1Xform operator* (const cc1Xform& xform) const;`

  Return the transformation object that is the result of composing this transformation object with *xform*, that is, the transformation object that gives the same result as mapping a point first by this transformation object and then by *xform*.

  Parameters

xform	The transformation object to compose with this transformation object.

- `double operator* (double pt) const;`

  Maps the point pt according to the transformation object:

  ```
  result = scale * pt + offset;
  ```

  Parameters

pt	The point to be mapped.

operator==

```
bool operator== (const cc1Xform& that) const;
```

Return true if this transformation object is equal to *that*.

Parameters

that	The other transformation object.

operator!=

```
bool operator!= (const cc1Xform&) const;
```

Return true if this transformation object is not equal to *that*.

Parameters

that	The other transformation object.

Public Member Functions

offset

```
double offset();
void offset(double offset);
```

- `double offset();`

  Returns the offset component of the transformation object. The offset is the amount that the transformation matrix adds or subtracts to a scaled point.

- `void offset(double offset);`

  Sets the offset of the transformation object.

  Parameters

  offset The amount to add or subtract to a point.

scale

```
double scale();
void scale(double scale);
```

- `double scale();`

  Returns the scale component of the transformation object. The scale is the amount by which the transformation object multiplies a point.

- `void scale(double scale);`

  Parameters

  scale The amount by which to multiply a point.

inverse

```
cc1Xform inverse() const;
```

Return the inverse transformation object for this transformation object.

Throws

ccMathError::Singular
The transformation is singular (scale = 0).

cc1Xform

compose `cc1Xform compose(const cc1Xform& xform) const;`

Return the transformation object that is the result of composing this transformation object with *xform*, that is, the transformation object that gives the same result as mapping a point first by this transformation object and then by *xform*.

Parameters

xform The transformation object to compose with this transformation object.

Notes

This is the same function as the multiplication (*) operator when the other operator is a **cc1Xform**.

mapPoint `double mapPoint (double pt) const;`

Map the point *pt* according to the transformation object:

```
result = scale * pt + offset;
```

Parameters

pt The point to be mapped.

Notes

This is the same function as the multiplication (*) operator when the other operator is a point.

invMapPoint `double invMapPoint (double pt) const;`

Maps the point *pt* by the inverse of this transformation object:

```
result = 1/scale * (pt - offset);
```

Throws

ccMathError::Singular
The transformation is singular.

mapVector `double mapVector (double vect) const;`

Scales the vector:

```
result = scale * vect;
```

Parameters

vect The vector to scale.

invMapVector

```
double invMapVector (double vect) const;
```

Scales the vector: result = 1/scale * vect

Parameters

vect

Throws

ccMathError::Singular if the transformation object is singular.

isIdentity

```
bool isIdentity() const;
```

Return true if the transformation object is the identity transform.

isSingular

```
bool isSingular();
```

Return true if the transformation is singular; that is, all points map to the same point.

Data Members

I

```
static const cc1Xform I;
```

The identity transformation object for **cc1Xform**.

cc1Xform

cc2Matrix

```
#include <ch_cvl/matrix.h>

template <int D> class ccMatrix;

typedef ccMatrix<2> cc2Matrix;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class implements a four element, 2 by 2 square matrix that allows for 2D transformations.

Note The template parameter, D, in the class definition specifies the dimension of a **ccMatrix**. For example, **ccMatrix<2>** specifies a 2x2 matrix. **cc2Matrix** is a type definition for a **ccMatrix<>** object instantiated with a template parameter of 2. Similarly, **ccMatrix<3>** specifies a 3x3 matrix. **cc3Matrix** is a type definition for a **ccMatrix<>** object instantiated with a template parameter of 3.

The **cc2Matrix** member functions let you specify the elements of the transformation matrix in three different modalities:

1. The explicit mode. In this modality you explicitly define the four elements of the matrix. When you use this modality, the mapping between image coordinates (I_x, I_y) and client coordinates (C_x, C_y) is:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix}$$

   where e_{11}, e_{12}, e_{21}, e_{22} are the matrix elements that you have specified. (For the definition of image coordinates and client coordinates as well as an overview on image transformations see *Images and Coordinates* in the *CVL User's Guide*).

2. The scale-rotation mode. In this modality the matrix is specified by the rotation of the x-axis (r_x), the rotation of the y-axis (r_y), the x-scale (s_x) and the y-scale (s_y). When you use this modality, the mapping between image coordinates (I_x, I_y) and client coordinates (C_x, C_y) is:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} s_x \cos r_x & -s_y \sin r_y \\ s_x \sin r_x & s_y \cos r_y \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix}$$

3. The shear-aspect mode. In this modality the matrix is specified by the scale of the coordinate system (S), the aspect ratio of the x- and y- axes (A), the shear angle (K) and the rotation angle of the coordinate system(R).When you use this modality, the mapping between image coordinates (I_x, I_y) and client coordinates (C_x, C_y) is:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} S\cos R & AS(-\sin R - \cos R \tan K) \\ S\sin R & AS(-\cos R - \sin R \tan K) \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix}$$

The scale-rotation and shear-aspect modalities are equivalent, so you can use the one that best suits your application or existing algorithms. However, it is important not mix elements from one modality with elements from the other modality. For example, if you specified a shear angle (K) in the shear-aspect mode, then $A \neq s_y / s_x$. In general the scale-rotation mode is easier to work with.

Note **cc2Matrix** is the name of the class as it is used throughout the Cognex Vision Library. This name is actually a **typedef** for **ccMatrix<2>.** Both forms are valid and entirely equivalent.

Constructors/Destructors

cc2Matrix

```
cc2Matrix();

cc2Matrix(double e11, double e12, double e21, double e22);

cc2Matrix(const ccRadian& xRot, const ccRadian& yRot,
  double xScale, double yScale);

cc2Matrix(double scale, double aspect,
  const ccRadian& shear, const ccRadian& rotation);
```

- `cc2Matrix();`

  The default constructor initializes this matrix to the identity matrix ($\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$)

- `cc2Matrix(double e11, double e12, double e21, double e22);`

  Initializes this matrix according to the explicit mode. The constructor sets the elements of the matrix to *e11*, *e12*, *e21*, *e22*.

  Parameters

e11	The element of the matrix corresponding to the first row and first column.
e12	The element of the matrix corresponding to the first row and second column.
e21	The element of the matrix corresponding to the second row and first column.
e22	The element of the matrix corresponding to the second row and second column.

- `cc2Matrix(const ccRadian& xRot, const ccRadian& yRot, double xScale, double yScale);`

  Initializes this matrix according to the scale-rotation mode.

xRot	The x-rotation; the amount by which to rotate the x-axis in radians.
yRot	The y-rotation; the amount by which to rotate the y-axis in radians.
xScale	The x-scale; the amount by which to scale the x-axis units.
yScale	The y-scale; the amount by which to scale the y-axis units.

- `cc2Matrix(double scale, double aspect, const ccRadian& shear, const ccRadian& rotation);`

Initializes this matrix according to the shear-aspect mode.

scale	The scale element; the amount by which to scale all units.
aspect	The aspect ratio; the ratio of the x-axis units to the y-axis units.
shear	The shear element; the amount by which to rotate the y-axis in radians.
rotation	The rotation element; the amount by which to rotate the entire coordinate system in radians.

Operators

operator+

```
cc2Matrix operator+(const cc2Matrix& m) const;
```

Returns the sum of this matrix and the matrix *m*.

Parameters

m	The matrix added to this matrix.

operator+=

```
cc2Matrix& operator+=(const cc2Matrix& m);
```

Sets this matrix to the sum of this matrix and the matrix *m*.

Parameters

m	The matrix added to this matrix.

operator-

```
cc2Matrix operator-() const;
cc2Matrix operator-(const cc2Matrix& m) const;
```

- `cc2Matrix operator-() const;`

Returns a matrix whose elements are the opposite (i.e. multiplied by -1) of the elements of this matrix.

- `cc2Matrix operator-(const cc2Matrix& m) const;`

Returns the difference between this matrix and the matrix *m*.

Parameters

m	The matrix subtracted from this matrix.

operator-=

```
cc2Matrix& operator-=(const cc2Matrix& m);
```

Sets this matrix to the difference of this matrix and the matrix *m*.

Parameters

m	The matrix subtracted from this matrix.

operator*

```
cc2Matrix operator*(double s) const;
cc2Matrix operator*(const cc2Matrix& m) const;
cc2Vect operator*(const cc2Vect& v);
```

- `cc2Matrix operator*(double s) const;`

  Returns a matrix whose elements are equal to the elements of this matrix times *s*.

  Parameters

s	The value each element of this matrix is multiplied by.

- `cc2Matrix operator*(const cc2Matrix& m) const;`

  Returns the matrix that is the composition of this matrix and the matrix *m*. Mapping a point with the resulting matrix is equivalent to first mapping the point by the matrix *m* and then by this matrix.

  Parameters

m	The matrix composed with this matrix.

- `cc2Vect operator*(const cc2Vect& v) const;`

  Returns the vector resulting from the product of this matrix and the vector *v* (representing a point). The vector returned is the point *v* transformed by this matrix.

  Parameters

v	The vector transformed by this matrix.

operator*=

```
cc2Matrix& operator*=(double s);
cc2Matrix& operator*=(const cc2Matrix& m);
```

- `cc2Matrix& operator*=(double s);`

  Sets this matrix to the product of this matrix and *s*.

Parameters

s The value each element of this matrix is multiplied by.

- `cc2Matrix& operator*=(const cc2Matrix& m);`

Sets this matrix to the composition of this matrix and *m*.

Parameters

m The matrix composed with this matrix

operator/

```
cc2Matrix operator/(double s);
cc2Matrix operator/(const cc2Matrix& m);
```

- `cc2Matrix operator/(double s);`

Returns a matrix whose elements are equal to the elements of this matrix divided by s.

Parameters

s The value each element of this matrix is divided by.

- `cc2Matrix operator/(const cc2Matrix& m);`

Returns the matrix composed by this matrix and the inverse of *m*.

Parameters

m The matrix whose inverse is composed with this matrix.

operator/=

```
cc2Matrix& operator/=(double s);
cc2Matrix operator/=(const cc2Matrix& m);
```

- cc2Matrix& operator/=(double s);

Sets this matrix to the quotient of this matrix and *s*.

Parameters

s The value each element of the matrix is divided by.

- `cc2Matrix operator/=(const cc2Matrix& m);`

Sets this matrix to the matrix composed by this matrix and the inverse of *m*.

Parameters

m The matrix whose inverse is composed with this matrix.

Throws

ccMathError::Singular
The matrix used as a divisor is singular.

operator==

```
bool operator==(const cc2Matrix& m) const;
```

Returns true if all the elements of this matrix are the same as the corresponding elements of the matrix *m*.

m The matrix compared to this matrix.

operator!=

```
bool operator!=(const cc2Matrix& m) const;
```

Returns true if this matrix is not the same as *m*.

Parameters

m The matrix compared to this matrix.

Friends

operator *=

```
friend cc2Vect operator*=(const cc2Vect& v,
  const cc2Matrix& m);
```

Returns the vector resulting from the left-product of the matrix *m* with the vector *v*. If

$v = \begin{bmatrix} V_x \\ V_y \end{bmatrix}$ and $m = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$ the left-product of *m* with *v* is defined as

$$\begin{bmatrix} V_x & V_y \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} (V_x m_{11} + V_y m_{21}) & (V_x m_{12} + V_y m_{22}) \end{bmatrix}$$

Parameters

v The vector multiplied with *m*.

m The matrix multiplied with v.

Public Member Functions

determinant

```
double determinant() const;
```

Returns the determinant of this matrix. If ***this*** $= \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}$ the value returned is $t_{11}t_{22} - t_{12}t_{21}$.

inverse

```
cc2Matrix inverse() const;
```

Returns the inverse of this matrix. The inverse of this matrix is $\left(\frac{1}{det[this]}\right)\begin{bmatrix} t_{22} & -t_{12} \\ -t_{21} & t_{11} \end{bmatrix}$ where $det[this]$ is the determinant of this matrix.

Throws

ccMathError::Singular
The transformation matrix is singular.

transpose

```
cc2Matrix transpose() const;
```

Returns the transpose of this matrix. If $this = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}$ the transpose of this matrix is $\begin{bmatrix} t_{11} & t_{21} \\ t_{12} & t_{22} \end{bmatrix}$.

isSingular

```
bool isSingular() const;
```

Returns true if this matrix is singular (i.e. if $det[this]$ is exactly equal to zero).

isIdentity

```
bool isIdentity() const;
```

Returns true if this matrix is the identity matrix.

element

```
double element(c_Int32 row,c_Int32 column) const;
void element(c_Int32 row,c_Int32 column, double value);
```

- `double element(c_Int32 row,c_Int32 column) const;`

Returns the element of this matrix specified by the indices *row* and *column*.

Parameters

row — The row-index.

0: Identifies the first row.

1: Identifies the second row.

column — The column-index.

0: Identifies the first column.

1: Identifies the second column.

- `void element(c_Int32 row,c_Int32 column, double value);`

  Sets the element of this matrix specified by the indices *row* and *column* to *s*.

  Parameters

  row — The row-index.

  0: identifies the first row.

  1: identifies the second row.

  column — The column-index.

  0: identifies the first column.

  1: identifies the second column.

  value — The value the matrix element specified by *row* and *column* is set to.

xRot `ccRadian xRot() const;`

Returns the x-rotation element of this matrix when using the scale-rotation specification.

Throws

ccMathError::Singular — This matrix is singular.

yRot `ccRadian yRot() const;`

Returns the y-rotation element of this matrix when using the scale-rotation specification.

Throws

ccMathError::Singular — This matrix is singular.

xScale `double xScale() const;`

Returns the x-scale element of this matrix when using the scale-rotation specification.

Throws

ccMathError::Singular — This matrix is singular.

yScale `double yScale() const;`

Returns the y-scale element of this matrix when using the scale-rotation specification.

Throws
ccMathError::Singular
This matrix is singular.

scale `double scale() const;`

Returns the scale element of this matrix when using the shear-aspect specification.

Throws
ccMathError::Singular
This matrix is singular.

aspect `double aspect() const;`

Returns the aspect element of this matrix when using the shear-aspect specification.

Throws
ccMathError::Singular
This matrix is singular.

shear `ccRadian shear() const;`

Returns the shear element of this matrix when using the shear-aspect specification.

Throws
ccMathError::Singular
This matrix is singular.

rotation `ccRadian rotation() const;`

Returns the rotation element of this matrix when using the shear-aspect specification.

Throws
ccMathError::Singular
This matrix is singular.

Data Members

I `static const cc2Matrix I;`

The identity transformation matrix for **cc2Matrix.**

cc2Point

```
#include <ch_cvl/shapes.h>

class cc2Point : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a point. In most cases, you can use either **cc2Point** or **cc2Vect** to describe locations in a coordinate system. This class, however, is a full-fledged **ccShape** andinherits all of the methods of that class.

Constructors/Destructors

cc2Point

```
cc2Point();

cc2Point(const cc2Vect& v);

cc2Point(double x, double y);
```

- `cc2Point();`

  Default constructor. Constructs a point with coordinates (0,0).

- `cc2Point(const cc2Vect& v);`

  Conversion constructor. Constructs a point with coordinates equal to the coordinates of the supplied vector.

  Parameters

  v The vector.

- `cc2Point(double x, double y);`

  Constructs a point with *x* and *y* coordinates equal to the supplied values.

  Parameters

  x The x value.

y The y value.

Operators

operator==

```
bool operator==(const cc2Point &rhs) const;
```

Returns true if this point is equal to *rhs*, under strict floating point equality.

Parameters

rhs The other **cc2Point**.

operator!=

```
bool operator!=(const cc2Point &rhs) const;
```

Returns true if this point is not equal to *rhs*, under strict floating point equality.

Parameters

rhs The other **cc2Point**.

Public Member Functions

x

```
double x() const;
double x(double val);
```

- ```
  double x() const;
  ```

  Gets the x coordinate of this point.

- ```
  double x(double val);
  ```

  Sets the x coordinate of this point.

  Parameters

  val The value to which the x coordinate is set.

y

```
double y() const;
void y(double val);
```

- ```
  double y() const;
  ```

  Gets the y coordinate of this point.
  ```

- `void y(double val);`

  Sets the y coordinate of this point.

  Parameters

  val The value to which the y coordinate is set.

vect

```
const cc2Vect& vect() const;
void vect(const cc2Vect &val);
```

- `const cc2Vect& vect() const;`

  Returns the vector representation of this point.

- `void vect(const cc2Vect &val);`

  Sets the vector representation of this point.

  Parameters

  val The vector that represents this point.

map

```
cc2Point map(const cc2Xform& c) const;
```

Returns the result of mapping this point with the transformation object *c*.

Parameters

c The transformation object.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this point.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For points, this function always returns true. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For points, this function always returns false. See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

For points, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For points, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

For points, this function always returns false. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For points, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible `virtual bool isReversible() const;`

For points, this function always returns true. See **ccShape::reverse()** for more information.

boundingBox `virtual ccRect boundingBox() const;`

Returns the smallest rectangle that encloses this point.

Notes

This function succeeds the deprecated **encloseRect()**. However, note that **boundingBox()** returns a rectangle of size (0, 0), while **encloseRect()** returned a rectangle of size (1, 1) for points.

See **ccShape::boundingBox()** for more information.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const;`

Returns this **cc2Point**.

Parameters

p	The point.

perimeter

```
virtual double perimeter() const;
```

Returns 0.0 for any point.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this shape to the given point. For a point, the returned perimeter position specifies the point itself.

Parameters

info Shape information for this point.

point The other point.

Notes

The **ccPerimPos::distAlongPrimitive()** component of the returned perimeter position is equal to 0.0.

See **ccShape::nearestPerimPos()** for more information.

startPoint

```
virtual cc2Vect startPoint() const;
```

Returns this **cc2Point**. See **ccShape::startPoint()** for more information.

endPoint

```
virtual cc2Vect endPoint() const;
```

Returns this **cc2Point**. See **ccShape::endPoint()** for more information.

startAngle

```
virtual ccRadian startAngle() const;
```

Throws

ccShapesError::NoTangent
hasTangent() is always false for **cc2Point**.

See **ccShape::startAngle()** for more information.

endAngle

```
virtual ccRadian endAngle() const;
```

Throws

ccShapesError::NoTangent
hasTangent() is always false for **cc2Point**.

See **ccShape::endAngle()** for more information.

cc2Point

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Throws

ccShapesError::NoTangent
hasTangent() is always false for **cc2Point**.

See **ccShape::tangentRotation()** for more information.

windingAngle

```
virtual ccRadian windingAngle(const cc2Vect &p) const;
```

Returns 0.0 for a **cc2Point**.

Parameters

p	The start point of the vector *p->t* where *t* is this point.

See **ccShape::windingAngle()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns this **cc2Point**. See **ccShape::reverse()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Adds this **cc2Point** to *result* if **params.computeTangents()** is false.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position chains are appended.

Notes

If **params.computeTangents()** is true, this **cc2Point** does not contribute any samples to *result*.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this point mapped by *X*.

Parameters

X	The transformation object.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a zero-length line segment with both endpoints coincident with this point. See **ccShape::decompose()** for more information.

subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to this point.

Parameters

info	Shape information for this point.
range	The perimeter range. If the distance component of this **ccPerimRange** is greater than 0.0, it is internally clipped to 0.0.

See **ccShape::perimeter()** for more information.

cc2Point

cc2Rigid

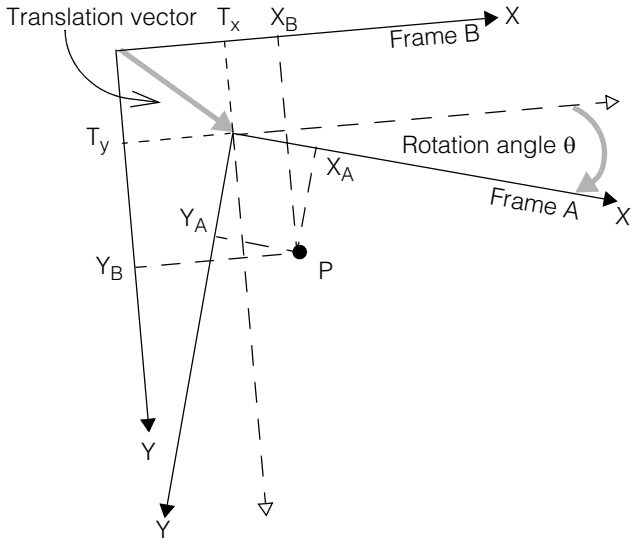
```
#include <ch_cvl/xform.h>

class cc2Rigid;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

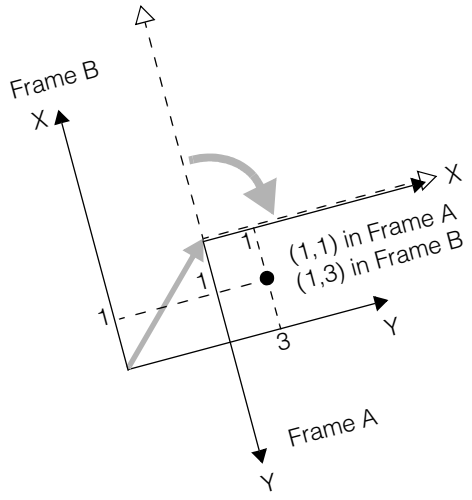
This class implements a rigid transformation of coordinate data.The transformation can be also thought of as a mapping of features from an initial coordinate frame to a second frame. The transformation is called rigid because the data are mapped by a simple rotation and translation, no distortions due to scaling or skews are present. The following figure illustrates a rigid transformation that maps features from Frame A to Frame B. The transformation is defined by a 2-element translation vector and by a rotation angle. The elements of the translation vector specify the amount by which Frame A is translated along the X- and Y- directions of Frame B (T_x and T_y in the following figure), while the rotation angle specifies the angle by which the coordinate axes are rotated from Frame B (θ in the following figure)



If your data are initially represented in Frame A, the member functions of **cc2Rigid** allow you to map them to their representation in Frame B. For example, if (X_A, Y_A) are the coordinates of the point P in Frame A, the class provides you with ways to return the coordinates of the point in Frame B (X_B, Y_B). The two points are mathematically related by the following expression:

$$\begin{bmatrix} X_B \\ Y_B \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X_A \\ Y_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

As an illustration, consider the rigid transformation characterized by the translation vector (2,2) and the rotation angle of 90 degrees (see following figure). In this case the point (1,1) in Frame A is mapped to the point (1, 3) in Frame B.



$$\begin{bmatrix} X_B \\ Y_B \end{bmatrix} = \begin{bmatrix} \cos(90) & -\sin(90) \\ \sin(90) & \cos(90) \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Constructors/Destructors.

cc2Rigid

```
cc2Rigid();
cc2Rigid(const ccDegree& a, const cc2Vect& t);
```

- `cc2Rigid();`

  The default constructor initializes this rigid transformation to the identity transformation, that is the transformation which maps features to their current location, leaving them unchanged. The identity transformation is defined by a translation vector whose components are both *0* (no translation) and a rotation angle of *0* degrees (no rotation).

- `cc2Rigid(const ccDegree& a, const cc2Vect& t);`

  Initializes the rotation angle and the translation vector of this rigid transformation.

  Parameters

a	The rotation angle assigned to this rigid transformation.
t	The translation vector assigned to this rigid transformation.

Operators

operator*

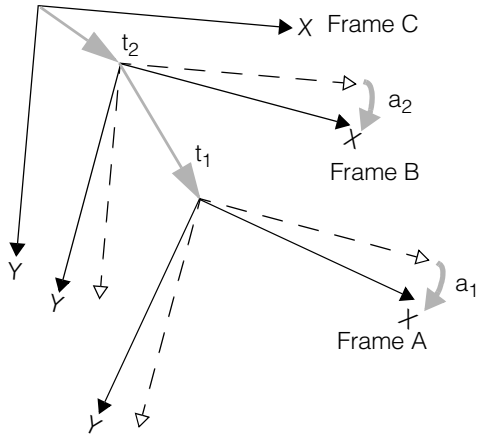
```
cc2Rigid operator*(const cc2Rigid& xform) const;
cc2Xform operator*(const cc2Xform& xform) const;
cc2Vect operator*(const cc2Vect &pt) const;
```

- `cc2Rigid operator*(const cc2Rigid& xform) const;`

  Returns the rigid transformation that is the composition of this rigid transformation and *xform*. Mapping the resulting rigid transformation is equivalent to first mapping *xform* and then this rigid transformation. For example, assume that *xform* is defined by the translation vector t_1 and by the rotation angle a_1 while this rigid transformation is defined by the translation vector t_2 and the rotation angle a_2 (see following figure). The rigid transformation returned by the operator has the same effect as the following sequence of transformations.

  1. *xform* maps data from Frame A to Frame B.

2. This rigid transformation maps data from Frame B to Frame C.



Parameters

xform The rigid transformation composed with this rigid transformation.

- `cc2Xform operator*(const cc2Xform& xform) const;`

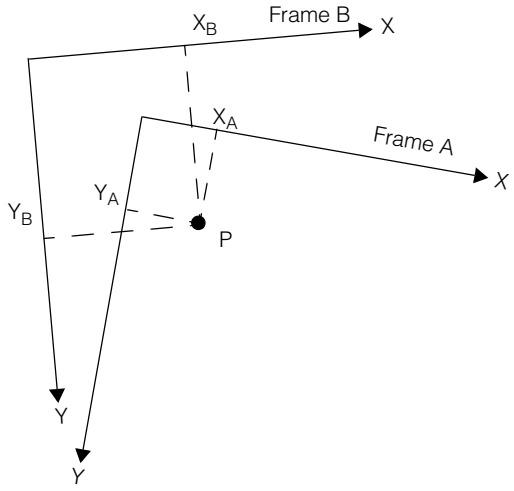
This operator returns the **cc2Xform** transformation object resulting from the composition of this rigid transformation and *xform*. Mapping a point with the resulting **cc2Xform** transformation object is equivalent to first mapping the point by *xform* and then by this rigid transformation.

Parameters

xform The **cc2Xform** transformation object composed with this rigid transformation.

- `cc2Vect operator*(const cc2Vect &pt) const;`

  Maps the coordinates of the point *pt* using this rigid transformation. The point (X_A, Y_A) in Frame A is mapped to (X_B, Y_B) in Frame B (see following figure). The operation performed by this operator is equivalent to the one performed by the function **mapPoint**.



Parameters

pt The point whose coordinates are mapped.

operator == `bool operator==(const cc2Rigid& xform) const;`

Returns true if this rigid transformation is exactly the same as *xform* (that is, if the translation and rotation of the two transformations are the same).

Parameters

xform The rigid transformation compared to this rigid transformation.

operator!= `bool operator!=(const cc2Rigid& xform) const;`

Returns true if this rigid transformation is not the same as *xform*.

Parameters

xform The transformation compared to this rigid transformation.

Friends

operator*

```
cc2Xform operator*(const cc2Xform &x,
  const cc2Rigid &r);
```

Returns the **cc2Xform** resulting from the composition of the **cc2Xform** *x* and the rigid transformation *r*. Mapping a point with the resulting **cc2Xform** transformation object is equivalent to first mapping the point by *r* and then by *x*.

Parameters

x	The **cc2Xform** transformation object to be composed with the rigid transformation *r*.
r	The rigid transformation to be composed with the **cc2Xform** transformation object *x*.

Public Member Functions

angle

```
const ccDegree& angle()const;

void angle(const ccDegree& a);
```

- `const ccDegree& angle() const;`

  Returns the rotation angle of this rigid transformation in degrees.

- `void angle(const ccDegree& a);`

  Sets the rotation angle of this rigid transformation to *a*.

  Parameters

a	The rotation angle assigned to this rigid transformation.

cosAngle

```
double cosAngle()const;
```

Returns the cosine of the rotation angle of this rigid transformation.

sinAngle

```
double sinAngle()const;
```

Returns the sine of the rotation angle of this rigid transformation.

trans

```
const cc2Vect& trans()const;
void trans(const cc2Vect& t);
```

- `const cc2Vect& trans()const;`

  Returns the translation vector of this rigid transformation.

- `void trans(const cc2Vect& t);`

  Sets the translation vector of this rigid transformation to *t*.

  Parameters

t	The translation vector assigned to this rigid transformation.

linear

```
cc2Xform linear() const;
```

Returns the equivalent **cc2Xform** form of this rigid transformation.

compose

```
cc2Rigid compose(const cc2Rigid& xform) const;
cc2Xform compose(const cc2Xform& xform) const;
```

- `cc2Rigid compose(const cc2Rigid& xform)const;`

  Returns the composition of this rigid transformation with *xform*. This function implements the same operation as **operator *(const cc2Rigid& xform)**.

  Parameters

xform	The rigid transformation composed with this transformation.

- `cc2Xform compose(const cc2Xform& xform)const;`

  Returns the composition of this rigid transformation with the **cc2Xform** transformation *xform*. This function implements the same operation as **operator *(const cc2Xform& xform)**.

  Parameters

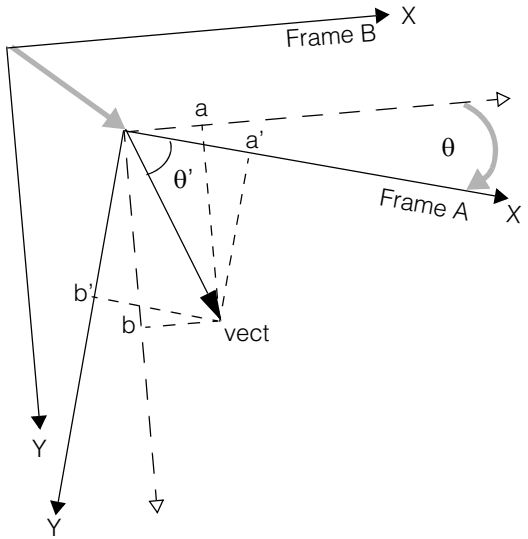
xform	The **cc2Xform** transformation composed with this transformation.

inverse `cc2Rigid inverse() const;`

Returns the inverse of this rigid transformation. The inverse of this rigid transformation is defined as the rigid transformation that, when composed with this rigid transformation, returns the identity transformation. If this rigid transformation maps points from Frame A to Frame B, the inverse of this rigid transformation maps points from Frame B back to Frame A.

mapVector `cc2Vect mapVector (const cc2Vect &vect) const;`

Maps the components of *vect* by this rigid transformation. If (a', b') are the components of *vect* in Frame A and (a, b) are the components of *vect* in Frame B, **mapVector** maps (a', b') to (a, b).



Parameters

vect The vector whose components are mapped by **mapVector**.

Notes.

Although a vector and a point are represented by the same data type (**cc2Vect**) it is important to keep in mind that they represent two distinct notions. A point represents a specific position in the coordinate space. A vector represents a length and direction in the same space but has no fixed location. A vector is depicted in drawings as an arrow. Points and vectors can be represented by the same class because a vector whose tail is placed at the origin specifies a unique point (at the end of the arrow).

invMapVector `cc2Vect invMapVector (const cc2Vect &vect) const;`

Implements the inverse of the mapping operated by **MapVector**. If we refer to the preceding figure, **invMapVector** maps (a, b) (in Frame B), to (a', b') in Frame A.

Parameters

vect The vector mapped by **invMapVector**.

mapAngle `ccRadian mapAngle (const ccRadian& ang) const;`

Maps the polar angle of some feature in coordinate Frame A (see preceding figure) to the polar angle in coordinate Frame B. If θ' is the polar angle of *vect* in Frame A, **mapAngle** maps θ' to ($\theta + \theta$').

Parameters

ang The polar angle of a feature in coordinate Frame A.

invMapAngle `ccRadian invMapAngle (const ccRadian& ang) const;`

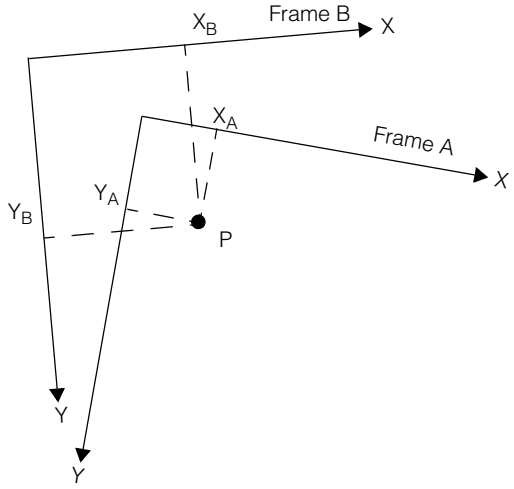
Implements the inverse of the mapping operated by **mapAngle**. If we refer to the preceding figure, **invMapAngle** maps ($\theta + \theta$') (the polar angle of *vect* in Frame B) to θ' (the polar angle of *vect* in Frame A).

Parameters

ang The polar angle of a feature in coordinate Frame B ($(\theta + \theta')$ in the preceding figure).

mapPoint `cc2Vect mapPoint (const cc2Vect &pt) const;`

Maps the point *pt* from its coordinates in Frame A (see following figure) to its coordinates in Frame B. This function implements the same operation as **operator *(const cc2Vect &pt)**. If (x_A, y_A) are the coordinates of *pt* in Frame A and (x_B, y_B) are the coordinates of *pt* in Frame B, **mapPoint** performs the following mapping: (x_A, y_A) -> (x_B, y_B).



Parameters

pt The point mapped by this transformation.

invMapPoint `cc2Vect invMapPoint (const cc2Vect &pt) const;`

Implements the inverse of the mapping operated by **mapPoint**. In the previous figure, **invMapPoint** maps (x_B, y_B) (the coordinates of *pt* in Frame B) to (x_A, y_A) (the coordinates of *pt* in Frame A).

Parameters

pt The point mapped by the inverse of this transformation.

isIdentity `bool isIdentity() const;`

Returns true if this rigid transformation is the identity transformation.

Data Members

```
static const cc2Rigid I;
```

The identity transformation for **cc2Rigid**.

cc2Rigid

cc2Vect

```
#include <ch_cvl/vector.h>

class cc2Vect;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Yes

This class describes a two-dimensional vector, a quantity that is determined by its length (magnitude) and direction. Throughout CVL, **cc2Vector** is used to describe points, so you can also think of a vector as an (x,y) coordinate pair.

Notes

cc2Vect is the name of the class as it is used throughout the Cognex Vision Library. This name is actually a **typedef** for **ccVector<2>**. Both forms are valid and entirely equivalent.

Do not confuse the CVL vector classes **cc2Vect** and **cc3Vect** with the C++ Standard Template Library's **vector** template class. Though both have the same mathematical underpinnings, CVL vector classes are generally used to describe the location of points while the STL **vector** class is used to implement flexible arrays.

Constructors/Destructors

cc2Vect

```
cc2Vect();
cc2Vect(double x, double y);
cc2Vect(double r, ccRadian t);
cc2Vect(const ccFPair& p);
cc2Vect(const ccDPair& p);
cc2Vect(const ccIPair& p);
```

- `cc2Vect();`

  The default constructor creates a vector in which both x and y components are set to zero.

- `cc2Vect(double x, double y);`

  Creates a vector with the specified x and y components.

  Parameters

x	The x-component of the vector.
y	The y-component of the vector.

- `cc2Vect(double r, ccRadian t);`

  Creates a vector from the specified radius and angle.

  Parameters

r	The radius of the circle.
t	The angle in radians. Angles are measured from the x-axis.

  Notes

  If the radius is zero, the angle information is not stored. The vector is a null vector.

- `cc2Vect(const ccFPair& p);`

  Creates a vector from the specified pair of **float**s.

  Parameters

p	The pair of **float**s.

- `cc2Vect(const ccDPair& p);`

  Creates a vector from the specified pair of **double**s.

  Parameters

p	The pair of **double**s.

- `cc2Vect(const ccIPair& p);`

  Creates a vector from the specified pair of integers (**c_Int32**).

  Parameters

p	The pair of integers.

Operators

operator[]

```
double& operator[](int d);
double operator[](int d) const;
```

- `double& operator[](int d);`

  Provides array-like access to the vector. Typically you use this form when your application works with both two- and three-dimensional vectors. This form allows you to set the value of a component.

  Parameters

  d The index of the element to return.

  0: The x-component.

  1: The y-component.

- `double operator[](int d) const;`

  Provides array-like access to the vector. Typically you use this form when your application works with both two- and three-dimensional vectors.

  Parameters

  d The index of the element to return.

  0: The x-component.

  1: The y-component.

operator+

```
cc2Vect operator+(const cc2Vect& v) const;
```

Returns a vector that is the result of adding this vector and another vector.

Parameters

v The vector to add to this vector.

operator+=

```
cc2Vect& operator+=(const cc2Vect& v);
```

Adds this vector to another vector and returns the result.

Parameters

v The vector to add to this vector.

operator-

```
cc2Vect operator-() const;
cc2Vect operator-(const cc2Vect&) const;
```

- `cc2Vect operator-() const;`

  The unary minus operator.

- `cc2Vect operator-(const cc2Vect& v) const;`

  Returns a vector that is the result of subtracting the vector v from this vector.

  Parameters

  v The vector to subtract from this vector.

operator-=

```
cc2Vect& operator-=(const cc2Vect& v);
```

Subtracts the vector v from this vector and returns the result.

Parameters

v The vector to subtract from this vector.

operator*

```
cc2Vect operator*(double d) const;
friend cc2Vect operator*(double, const cc2Vect&);
double operator*(const cc2Vect& v) const;
```

- `cc2Vect operator*(double d) const;`

  Returns a vector that is the result of multiplying each element in the vector by *d*.

  Parameters

  d The value by which to multiply each element of the vector.

- `friend cc2Vect operator*(double d, const cc2Vect& v);`

  Returns a vector that is the result of multiplying each element in the vector by *d*.

  Parameters

  d The value by which to multiply each element of the vector.

  v The vector.

- `double operator*(const cc2Vect& v) const;`

  Returns the dot product of this vector and the vector *v*.

  Parameters

v	The other vector.

operator*= `cc2Vect& operator*=(double d);`

Multiplies each element in this vector by *d* and returns the result.

Parameters

d	The value by which to multiply each element of the vector.

operator/ `cc2Vect operator/(double d) const;`

Returns a vector that is the result of dividing each element in this vector by *d*.

Parameters

d	The value by which to divide each element of the vector.

operator/= `cc2Vect& operator/=(double d) const;`

Divides each element in this vector by *d* and returns the result.

Parameters

d	The value by which to divide each element of the vector.

operator== `bool operator==(const cc2Vect& v) const;`

Returns true if this vector is equal to another vector.

Parameters

v	The other vector.

operator!= `bool operator!=(const cc2Vect&) const;`

Returns true if this vector is not equal to another vector.

Parameters

v	The other vector.

Public Member Functions

x

```
double x() const;
void x(double newX);
```

- `double x() const;`

  Returns the vector's x-component.

- `void x(double newX);`

  Sets the vector's x-component.

  Parameters

  newX The new x-component.

y

```
double y() const;
void y(double newY);
```

- `double y() const;`

  Returns the vector's y-component.

- `void y(double newY);`

  Sets the vector's y-component.

  Parameters

  newY The new y-component.

len

```
double len() const;
```

Returns the length (or magnitude) of the vector.

isNull

```
bool isNull() const;
```

Returns true if this is a null vector (both components are zero).

angle `ccRadian angle() const;`

Returns the vector angle in the range $-\pi <$ angle $<= \pi$. Angles are measured from the x-axis.

Throws

ccMathError::NullVector
Both the x-component and the y-component are zero.

unit `cc2Vect unit() const;`

Returns a unit vector parallel to this vector. A unit vector is a vector whose length is 1.

Throws

ccMathError::NullVector
Both the x-component and the y-component are zero.

project `cc2Vect project(const cc2Vect& v) const;`

Returns a vector that is the result of projecting the vector *v* onto this vector.

Parameters

v The vector to project onto this vector.

Throws

ccMathError::NullVector
Both the x-component and the y-component are zero.

An error is not thrown if *v* is null or the two vectors are perpendicular to each other.

perpendicular `cc2Vect perpendicular() const;`

Returns a vector perpendicular (rotated +90 degrees) to this vector. The new vector is the same length as this vector.

Notes

The null vector is considered perpendicular to itself

distance `double distance(const cc2Vect& v) const;`

Returns the distance to the other vector. This is the length of difference between this vector and vector *v*.

Parameters

v The other vector.

cc2Vect

dot

```
double dot(const cc2Vect& v) const;
```

Returns the dot product of this vector and the vector *v*.

Parameters

v	The other vector.

cross

```
double cross(const cc2Vect& v) const;
```

Returns the cross product of this vector and *v*.

Parameters

v	The other vector.

floor

```
ccIPair floor() const ;
```

Returns a **ccIPair** corresponding to **floor**(**cc2Vect::x()**), **floor(cc2Vect::y()**).

ceil

```
ccIPair ceil() const ;
```

Returns a **ccIPair** corresponding to **ceil**(**cc2Vect::x()**), **ceil(cc2Vect::y()**).

cc2Wireframe

```
#include <ch_cvl/wirefrm.h>

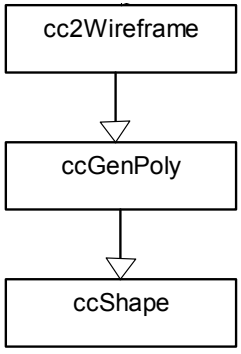
class cc2Wireframe : public ccGenPoly;
```

Class Properties

Copyable	Yes
Derivable	Not intended
Archiveable	Complex

The **cc2Wireframe** class is a general polygon (**ccGenPoly**) shape with an explicitly defined polarity value plus segment length tolerance information. The weight is implicitly defaulted to 1.0. However, the (deprecated) interface to PatMax allows you to specify weights that are applied to a vector of **cc2Wireframe** objects to be trained.

Note PatMax does not use the tolerance information in a **cc2Wireframe**.

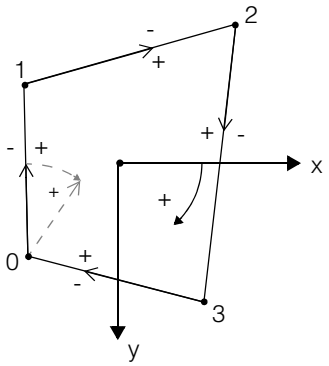


The above figure shows the **cc2Wireframe** class inheritance hierarchy.

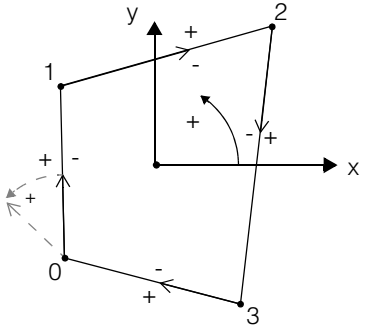
Polarity

Each wireframe has unique *polarity*. As you traverse the wireframe from one vertex to another, each segment has a positive side and a negative side. By default, the positive side of a segment with index N is that side to which the segment would move if it were to rotate in the direction of positive angle when the center of rotation is vertex N. The polarity is dependent on the direction of increasing indices, and the handedness of the coordinate system. Note that when the wireframe is closed, the inside of the wireframe will be of one polarity, and the outside will be of the opposite polarity.

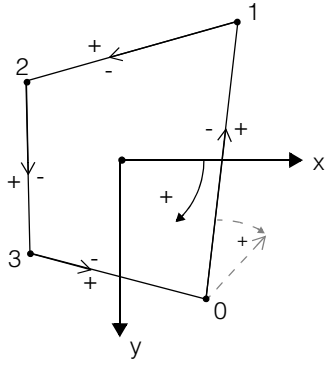
The following closed wireframe diagrams show the wireframe polarity for right-handed and left-handed coordinate systems.



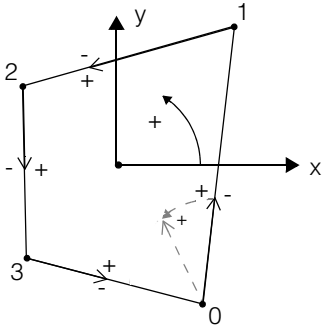
Left-handed coordinate system.
Positive rotation is clockwise.
Positive side of each segment on the right.



Right-handed coordinate system.
Positive rotation is counterclockwise.
Positive side of each segment on the left.

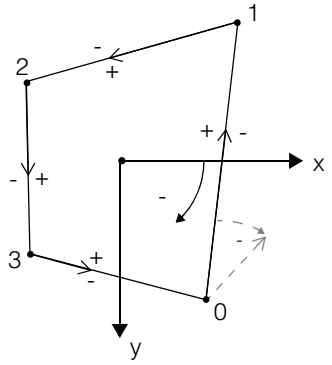


Left-handed coordinate system.
Positive rotation is clockwise.
Positive side of each segment on the right.



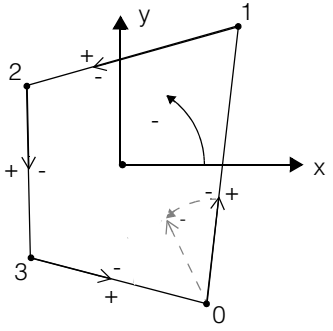
Right-handed coordinate system.
Positive rotation is counterclockwise.
Positive side of each segment on the left.

The diagrams above show the wireframe polarities for the given conditions. These polarities are the default state. Note that you can reverse the wireframe polarity to the *reversed* state by calling **cc2Wireframe::polarity()**. The two examples above are show below with reversed polarity.



Left-handed coordinate system.
Positive rotation is counter-clockwise.
Positive side of each segment on the left.

Reversed polarity



Right-handed coordinate system.
Positive rotation is clockwise.
Positive side of each segment on the right.

Each segment in the wireframe has associated tolerance information contained in a **ccDimTol** object that specifies *nominal*, *maximum,* and *minimum* segment lengths. A description of tolerances is presented in the **ccDimTol** reference page.

When tolerances are defined, the scale type associated with each segment indicates how tolerances change when the wireframe is scaled. The effect of scaling on tolerances is also discussed in the **ccDimTol** reference page.

Using PatMax with Wireframes

For new development, you should use **ccGenPoly** or **ccGenPolyModel** shapes rather than **cc2Wireframe** shapes when training PatMax on synthetic shapes. If you want to move to the new **train()** interface of PatMax but continue to use legacy **cc2Wireframe** shapes that may already exist in your code, you will need to put the wireframes into a shape tree prior to training. See the *Training with Wireframes* section of the *PatMax* chapter of the *CVL Vision Tools Guide* for sample code that demonstrates how to do this.

Constructors/Destructors

cc2Wireframe

```
cc2Wireframe() {};
cc2Wireframe(const ccGenPoly& gp);
cc2Wireframe(const ccGenPoly& gp, const ccDimTol & tol);
virtual ~cc2Wireframe();
```

- `cc2Wireframe() {};`

  Default constructor. Constructs a wireframe object with no vertices or segments.

- `cc2Wireframe(const ccGenPoly& gp);`

  Conversion constructor. Constructs a wireframe object that is equivalent to the supplied generalized polygon.

  Parameters

  gp — The generalized polygon used as a basis for the constructed wireframe object.

- `cc2Wireframe(const ccGenPoly& gp, const ccDimTol & tol);`

  Constructs a wireframe object that is equivalent to the supplied generalized polygon. All segments are set to use the specified segment length tolerance.

  Parameters

  gp — The generalized polygon used as a basis for the constructed wireframe object.

  tol — The segment length tolerance. The nominal tolerance dimension is ignored.

- `virtual ~cc2Wireframe();`

  Destructor. Destroys this wireframe shape.

Operators

operator==

```
bool operator==(const cc2Wireframe&) const;
```

This operator allows you to compare two **cc2Wireframe** objects. For example,

```
if(frame1 == frame2) {.....}
```

The equality expression evaluates to *true* if the objects are equal. It is *false* if they are not equal.

Parameters

cc2Wireframe The right-hand **cc2Wireframe** object. For example, *frame2*.

operator!=

```
bool operator!=(const cc2Wireframe&) const;
```

This operator allows you to compare two **cc2Wireframe** objects. For example,

```
if(frame1 != frame2) {.....}
```

The inequality expression evaluates to *true* if the objects are not equal. It is *false* if they are equal.

Parameters

cc2Wireframe The right-hand **cc2Wireframe** object. For example, *frame2*.

Public Member Functions

isPolarityReversed

```
bool isPolarityReversed() const;
```

Returns *true* if the polarity of the wireframe is reversed from its default state (see general polarity comments above), and false otherwise.

polarity

```
void polarity(bool isReversed);
```

If *isReversed* is true, the polarity of all segments in the wireframe is set to the reversed state. (See the wireframe polarity discussion starting on page 79). If *isReversed* is *false*, the polarity of all segments is set to the default state.

Parameters

isReversed The reverse polarity specifier, *true* or *false*.

Throws

ccShapesError::BadGeom
isMutable() is false.

isInsidePositive

```
bool isInsidePositive() const;
```

Returns *true* if the polarity of the wireframe is such that the inside of the wireframe is positive. Returns *false* otherwise.

Throws

ccShapesError::BadGeom
This wireframe is not closed.

insidePolarity

```
void insidePolarity(bool isPositive);
```

If *isPositive* is *true*, changes the polarity of all segments in this closed wireframe so that the inside of the wireframe is positive. If *isPositive* is *false*, sets the polarity of all segments so that the inside of the wireframe is negative.

Parameters

isPositive The wireframe polarity specifier; *true* or *false*.

Throws

ccShapesError::BadGeom
isMutable() is false.

segmentLengthTol

```
void segmentLengthTol(c_Int32 segmentIndex,
 const ccDimTol & tol);

ccDimTol segmentLengthTol(c_Int32 segmentIndex) const;
```

- ```
  void segmentLengthTol(c_Int32 segmentIndex,
   const ccDimTol & tol);
  ```

Sets the segment length tolerance for the segment with the specified index.

Parameters

segmentIndex The index for the target segment.

tol The new segment length tolerance. The nominal dimension is ignored.

Throws

ccShapesError::BadCoeff
segmentIndex is not a valid segment index.

ccShapesError::BadGeom
isMutable() is false, or the absolute tolerances specified are not possible.

The generalized polygon is unaffected in case of any throw.
  ```

- `ccDimTol segmentLengthTol(c_Int32 segmentIndex) const;`

  Returns the segment length tolerance for the segment with the specified index. The nominal segment length will reflect the actual length of the specified segment.

  Parameters

  segmentIndex The index for the target segment.

  Throws

  ccShapesError::BadCoeff
  If *segmentIndex* is not a valid segment index.

  The generalized polygon is unaffected in case of any throw.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this wireframe.

map

```
cc2Wireframe map(const cc2Rigid& c,
  double scale = 1.0) const;

cc2Wireframe map(const cc2Xform& c) const;
```

- `cc2Wireframe map(const cc2Rigid& c, double scale = 1.0) const;`

  Returns this wireframe scaled by *scale*, and mapped by *c*. The returned wireframe is mutable. For information on scaling of wireframe segment length tolerances see the **ccDimTol** class reference page.

  Parameters

  c The mapping transform used.

  scale The scale factor used.

  Throws

  ccShapesError::BadGeom
  If the scaled nominal segment length for any segment is not within the defined fixed tolerance ranges.

- `cc2Wireframe map(const cc2Xform& c) const;`

  Returns this wireframe mapped by *c*. Wireframe tolerances are scaled according to the scale type of each wireframe segment. Scaling is described in the **ccDimTol** class reference page. The scale factor for a given segment is defined as the ratio of the old segment length to the new segment length along a possibly elliptical arc.

  The returned generalized polygon is immutable unless **c.isIdentity()** is true.

  Parameters

  c The mapping transform used.

  Throws

  ccShapesError::BadGeom
  If the scaled nominal segment length for any segment is not within the defined fixed tolerance ranges.

insertVertex

```
void insertVertex(const cc2Vect & vertexPoint,
  double roundingSize = 0.0,
  const ccRadian& previousSegmentAngleSpan
    = ccRadian(0.0),
  const ccDimTol& previousSegmentLengthTol
    = ccDimTol (-1, -1, -1));

void insertVertex(c_Int32 previousVertexIndex,
  const cc2Vect& vertexPoint,
  double roundingSize = 0.0,
  const ccRadian& previousSegmentAngleSpan
    = ccRadian(0.0),
  const ccDimTol & previousSegmentLengthTol
    = ccDimTol (-1, -1, -1));
```

- ```
  void insertVertex(const cc2Vect & vertexPoint,
    double roundingSize = 0.0,
    const ccRadian & previousSegmentAngleSpan
      = ccRadian(0.0),
    const ccDimTol & previousSegmentLengthTol
      = ccDimTol (-1, -1, -1));
  ```

  Inserts the specified vertex after the last vertex in the generalized polygon. The vertex rounding, previous segment angle span, and previous segment length tolerance are also specified.

  Parameters

  vertexPoint The x,y location of the inserted vertex.

  roundingSize The vertex rounding size.
  ```

previousSegmentAngleSpan
The previous segment angle span.

previousSegmentLengthTol
The previous segment length tolerance.

An override, see the **ccGenPoly** base class.

- ```
  void insertVertex(c_Int32 previousVertexIndex,
    const cc2Vect& vertexPoint,
    double roundingSize = 0.0,
    const ccRadian & previousSegmentAngleSpan
    = ccRadian(0.0),
    const ccDimTol & previousSegmentLengthTol
      = ccDimTol (-1, -1, -1));
  ```

  Inserts the specified vertex after the vertex with index *previousVertexIndex*. The vertex rounding, previous segment angle span, and previous segment length tolerance are also specified.

  If *previousVertexIndex* = -1, the new vertex will become the first vertex.

  In all cases, vertex indices beyond the new vertex increase by 1 as a result of adding a new vertex.

  Parameters

  previousVertexIndex
  The index of the vertex prior to where the new vertex is inserted. A -1 indicates the new vertex should be the first vertex (index 0).

  vertexPoint The x,y location of the inserted vertex.

  roundingSize The vertex rounding size.

  previousSegmentAngleSpan
  The previous segment angle span.

  previousSegmentLengthTol
  The previous segment length tolerance.

  An override, see the **ccGenPoly** base class.
  ```

close

```
void close(
  const ccRadian & segmentAngleSpan = ccDegree(0),
  const ccDimTol& segmentLengthTol = ccDimTol(-1, -1, -1));
```

Closes the generalized polygon. It connects the vertex with the largest index to the vertex with the smallest index using a segment with the specified *segmentAngleSpan*. *segmentLengthTol* sets the new segment length tolerance.

There is no effect if the generalized polygon is already closed.

An override, see the **ccGenPoly** base class.

Parameters

segmentAngleSpan
: The segment angle span for the segment added when the polygon is closed.

segmentLengthTol
: The segment length tolerance.

Throws

ccShapesError::BadGeom
: If *segmentLengthTol* is not valid.

See **ccGenPoly** base class for more throw conditions.

cc2Xform

```
#include <ch_cvl/xform.h>

class cc2Xform;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a two-dimensional transformation object. The transformation object is made up of a four-element matrix and a two-element vector. Together, these two components allow for transformations along six degrees of freedom.

The **cc2Xform** member functions let you specify coordinate transformations two different ways. The difference between the two methods is the way you specify the elements of the four-element matrix. Both methods use the two-element vector for xy-translation.

The scale-rotation method for specifying transformation objects lets you set rotation of the x-axis, rotation of the y-axis, x-scale, and y-scale. The shear-aspect method lets you specify the scale of the coordinate system, the aspect ratio of the x to the y axis, the shear angle, and the rotation of the coordinate system. Both methods are equivalent, so you can use the one that suits your application or your existing algorithms. In general, the scale-rotation method is easier to use.

Note **cc2Xform** is the name of the class as it is used throughout the Cognex Vision Library. This name is actually a **typedef** for **ccXform<2>**. Both forms are valid and entirely equivalent.

Constructors/Destructors

cc2Xform

```
cc2Xform();

cc2Xform(const ccMatrix<2>& c, const cc2Vect& t);

cc2Xform(const cc2Vect& t, const ccRadian& xRot,
  const ccRadian& yRot, double xScale, double yScale);

cc2Xform(const cc2Vect& t, double scale, double aspect,
  const ccRadian& shear, const ccRadian& rotation);
```

- `cc2Xform();`

  The default constructor creates an identity transformation object.

- `cc2Xform(const ccMatrix<2>& c, const cc2Vect& t);`

  Creates a transformation object with a matrix component *c* and a translation component *t*.

  Parameters

c	The matrix component of the transformation object.
t	The translation component of the transformation object.

- `cc2Xform(const cc2Vect& t, const ccRadian& xRot, const ccRadian& yRot, double xScale, double yScale);`

  Creates a transformation object using the scale-rotation method.

  Parameters

t	The translation component; the amount by which to move the origin of the mapped coordinate system.
xRot	The x-rotation; the amount by which to rotate the x-axis in radians.
yRot	The y-rotation; the amount by which to rotate the y-axis in radians.
xScale	The x-scale; the amount by which to scale the x-axis units.
yScale	The y-scale; the amount by which to scale the y-axis unit.

- `cc2Xform(const cc2Vect& t, double scale, double aspect, const ccRadian& shear, const ccRadian& rotation);`

  Creates a transformation object using the shear-aspect method.

Parameters

t	The translation component; the amount by which to move the origin of the mapped coordinate system.
scale	The scale element; the amount by which to scale all units.
aspect	The aspect ratio; the ratio of the x-axis units to the y-axis units.
shear	The shear element; the amount by which to rotate the y-axis in radians.
rotation	The rotation element; the amount by which to rotate the entire coordinate system in radians.

Operators

operator*

```
cc2Xform operator* (const cc2Xform& xform) const;
cc2Vect operator* (const cc2Vect &pt) const;
```

- `cc2Xform operator* (const cc2Xform& xform) const;`

Returns the transformation object that is the composition of this transformation object and the transformation object *xform*. Mapping a point with the resulting transformation object has the same effect as mapping the point by the *xform* transformation object and then by this transformation object.

Parameters

xform	The transformation object to compose with this one.

Notes

Using this operator is the same as using the **compose()** function:

```
xform.compose(xform2) == xform * xform2
```

- `cc2Vect operator* (const cc2Vect &pt) const;`

Return the result of mapping the point *pt* with this transformation object. This is has the same effect as:

```
resultPt = matrix() * pt + trans();
```

Parameters

pt	The point to map.

Notes

This operator is the same as using the **mapPoint()** function:

```
xform.mapPoint(pt) == xform * pt
```

operator==

```
bool operator== (const cc2Xform& that) const;
```

Return true if this transformation object is the same as *that* transformation object.

Parameters

that The other transformation object to compare.

operator!=

```
bool operator!= (const cc2Xform&) const;
```

Parameters

that The other transformation object to compare.

Public Member Functions

matrix

```
const ccMatrix<2>& matrix();
void matrix(const ccMatrix<2>& c);
```

- `const ccMatrix<2>& matrix();`

  Returns the 2x2 matrix component of the transformation object.

- `void matrix(const ccMatrix<2>& c);`

  Sets the matrix component of the transformation object to *c*.

  Parameters

  c The matrix component of the transformation object.

trans

```
const cc2Vect& trans();
void trans(const cc2Vect& t);
```

- `const cc2Vect& trans();`

  Returns the 2-element translation vector of the transformation object.

- `void trans(const cc2Vect& t);`

  Sets the translation vector component of the transformation object to *t*.

Parameters

t The translation component of the transformation object.

inverse

```
cc2Xform inverse() const;
```

Returns the inverse of this transformation object.

Throws

ccMathError::Singular
The transformation is singular.

compose

```
cc2Xform compose (const cc2Xform& xform) const;
```

Returns the transformation object that is the composition of this transformation object and the transformation object *xform*. Mapping a point with the resulting transformation object has the same effect as mapping the point by *xform*, then by this transformation object.

Parameters

xform The transformation object to compose with this one.

Notes

Using this function is the same as using the multiplication operator:

```
xform.compose(xform2) == xform * xform2
```

xRot

```
ccRadian xRot() const;
```

Returns the x-rotation element of the transformation object using the scale-rotation specification.

Notes

Be careful when you mix elements from the scale-rotation specification with elements of the shear-aspect specification. For example, if you specify a value for **shear()** in the shear-aspect specification, **aspect() != yScale() / xScale()**.

Throws

ccMathError::Singular
The transformation is singular.

yRot

```
ccRadian yRot() const;
```

Returns the y-rotation element of the transformation object using the scale-rotation specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

xScale `double xScale() const;`

Returns the x-scale element of the transformation object using the scale-rotation specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

yScale `double yScale() const;`

Returns the y-scale element of the transformation object using the scale-rotation specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

scale `double scale() const;`

Returns the scale element of the transformation object using the shear-aspect specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

aspect `double aspect() const;`

Returns the aspect element of the transformation object using the shear-aspect specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

shear `ccRadian shear();`

Returns the shear element of the transformation object using the shear-aspect specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

rotation `ccRadian rotation() const;`

Returns the rotation element of the transformation object using the shear-aspect specification.

Notes
See note to **xRot()** on page 93.

Throws
ccMathError::Singular
The transformation is singular.

mapAngle `ccRadian mapAngle (const ccRadian& ang) const;`

Return the results of mapping the angle *ang* with this transformation object.

Parameters
ang The angle to map in radians.

invMapAngle `ccRadian invMapAngle (const ccRadian& ang) const;`

Return the results of mapping the angle *ang* with the inverse of this transformation object.

Parameters

ang The angle to map in radians.

Throws

ccMathError::Singular
The transformation is singular.

mapPoint

```
cc2Vect mapPoint (const cc2Vect &pt) const;
```

Return the result of mapping the point *pt* with this transformation object. This is has the same effect as:

```
resultPt = matrix() * pt + trans();
```

Parameters

pt The point to map.

Notes

Using this function is the same as using the multiplication operator:

```
xform.mapPoint(pt) == xform * pt
```

invMapPoint

```
cc2Vect invMapPoint (const cc2Vect &pt) const;
```

Return the result of mapping the point *pt* with the inverse of this transformation object. This is has the same effect as:

```
resultPt = matrix().inverse() * (pt - trans());
```

Parameters

pt The point to map.

Throws

ccMathError::Singular
The transformation is singular.

mapVector

```
cc2Vect mapVector (const cc2Vect &vect) const;
```

Rotates and scales the vector *vect* by the transformation object, ignoring translation:

```
result = matrix() * vect;
```

Parameters

vect The vector to be mapped.

invMapVector

```
cc2Vect invMapVector (const cc2Vect &vect) const;
```

Rotates and scales the vector *vect* by the inverse of the transformation object, ignoring translation:

```
result = matrix().inverse() * vect;
```

Parameters

vect The vector to be mapped.

Throws

ccMathError::Singular
The transformation is singular.

mapArea

```
double mapArea (double area) const;
```

Returns the area after mapping by the *xform*.

Parameters

area The area to be mapped.

Notes

This function computes the mapped area by first calculating the area of a unit square and then multiplying area by that value. You may wish to call this function with the area 1.0 to get a conversion constant you can use without the overhead of calling this function multiple times.

invMapArea

```
double invMapArea (double area) const;
```

Returns the area after mapping by the inverse of the *xform*.

Parameters

area The area to be mapped.

Throws

ccMathError::Singular
The transformation is singular.

isIdentity

```
bool isIdentity() const;
```

Returns true if this transformation object is the identity object: all points map to themselves.

isSingular

```
bool isSingular() const;
```

Returns true if the transformation is singular: all points map to the same line.

Data Members

I

```
static const cc2Xform I;
```

The identity transformation matrix for **cc2Xform**.

cc2XformBase

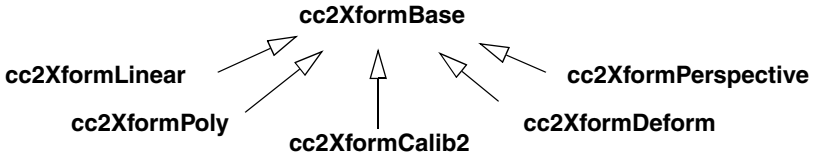
```
#include <ch_cvl/xfbase.h>

class cc2XformBase : public ccRepBase, public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This is the abstract base class for 2D coordinate transforms. From this base class the specialized classes such as **cc2XformPoly** and **cc2XformLinear** are derived.



Constructors/Destructors

Use the constructors of the derived classes.

Operators

operator*

```
cc2Vect operator*(const cc2Vect& pt) const;

cc2XformBasePtrh operator*(const cc2XformBase& rhs) const;
```

- `cc2Vect operator*(const cc2Vect& pt) const;`

  Returns the point *pt* mapped by this transformation. Using this operator is equivalent to using the **mapPoint** function.

  Parameters

pt	The point mapped by the operator.

Notes

Each derived class should make a **using** declaration of this operator. This assures that additional * operator functions overload, rather than hide, the base class operator.

- `cc2XformBasePtrh operator*(const cc2XformBase& rhs) const;`

Returns the composition of this transformation with the transformation *rhs*. Mapping a point with the resulting transformation is equivalent to first mapping the point by the transformation *rhs* and then by this transformation. The use of this operator is equivalent to the use of the **composeBase** function.

Parameters

rhs The transformation to be composed with this transformation.

Notes

The only allowed compositions are: **cc2XformLinear** * **cc2XformLinear**, **cc2XformLinear** * **cc2XformPoly**, **cc2XformPoly** * **cc2XformLinear**

Public Member Functions

mapPoint

```
virtual cc2Vect mapPoint(const cc2Vect& pt) const = 0;
```

Maps the point *pt* by using this transformation.

Parameters

pt The point mapped by this transformation.

isLinear

```
virtual bool isLinear() const = 0;
```

Returns true if this transformation is linear.

Notes

This function is designed to efficiently identify cases where transformations are guaranteed to be linear. The function may return false if it would take too much time to determine otherwise.

linearXform

```
virtual cc2XformLinear linearXform(const cc2Vect& atThisPt)
  const = 0;
```

Returns the linear transformation defined by the linear terms of the Taylor series expansion about the point *atThisPt.*

Parameters

atThisPt The point around which this transformation is linearized.

composeBase

```
virtual cc2XformBasePtrh composeBase(
  const cc2XformBase& rhs) const = 0;
```

Returns the composition of this transformation with the transformation *rhs*. Mapping a point with the resulting transformation is equivalent to first mapping the point by the transformation *rhs* and then by this transformation.

Parameters

rhs The transformation to be composed with this transformation.

Notes

The run-time type of the returned result is not guaranteed to be the same across different CVL releases, so do not write code that depends on the exact run-time type. For example, avoid using a dynamic cast.

It is expected that, for convenience, derived classes may declare functions such as **compose()** that take a particular transform type and return a particular transform type. The return value will not be heap allocated. For example;

```
cc2XformLinear: cc2XformLinear compose(
  const cc2XformLinear& rhs) const;
```

inverseBase

```
virtual cc2XformBasePtrh inverseBase() const = 0;
```

Returns a transformation that is the inverse of this transformation.

Throws

ccMathError::Singular

The transformation is singular and cannot be inverted.

Notes

In the case of nonlinear transformations, the inverse transformation may not be exact.

clone

```
virtual cc2XformBasePtrh clone() const = 0;
```

Returns a newly allocated copy of this transformation.

Notes

This function is required for polymorphic copying.

cc2XformBase

cc2XformCalib2

```
#include <ch_cvl/ccalib.h>

class cc2XformCalib2 : public cc2XformBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class contains a nonlinear calibration transform. It models the effects of perspective distortion and lens distortion. A general reference for terminology used in this description is *Multiple View Geometry in Computer Vision* by Richard Hartley and Andrew Zisserman.

A calibration transform C has the following form when mapping points from physical space to image space:

C = L composed with R_{rad} composed with P

where:

P is a perspective transform and depends on the position and orientation of the object plane with respect to the camera.

R_{rad} is a radial transform and depends on the camera lens used. This transform is modeled as a cubic polynomial transform using one distortion coefficient,

L is a linear transform and depends on camera properties such as focal length (scale), aspect, skew, and position of the optical axis relative to the image center. L does not contain any rotation.

Thus, P depends on the properties of the image acquisition system which are external to the camera. These are known as the *extrinsic* calibration parameters. R_{rad} and L depend on camera properties which are internal to the camera. These are known as *intrinsic* calibration parameters.

The calibration transform mapping image points to physical points is obtained by composition of inverses of L, R_{rad}, and P composed in reverse order.

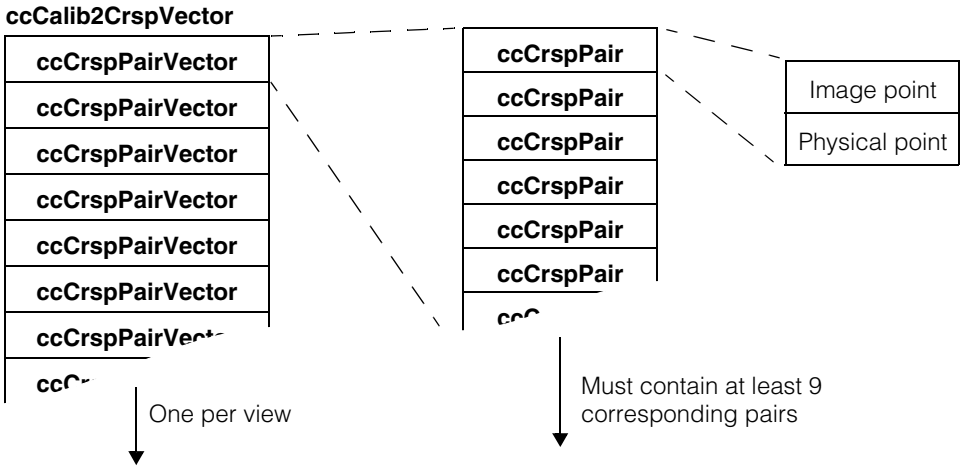
There are two ways to compute this transform:

1. Using known intrinsic and extrinsic calibration parameters.
2. Using correspondences between physical points and client points generated from one or more views. (Client space is usually the same as image space in the calibration image.)

Use **init()** to create the transform in one of these two ways.

Correspondences

The following figure describes how we manage corresponding image points and physical points. Each member of the **ccCalib2CrspVector** vector is a vector of corresponding image point - physical point pairs. The points in each **ccCrspPairVector** vector all come from one image (a view) and must contain at least 9 corresponding point pairs.



Note that the **cc2XformCalib2** is asymmetric between image coordinates and physical coordinates. **cc2XformCalib2** is designed to map from physical coordinates to image coordinates. You should always initialize a **cc2XformCalib2** object so that it maps from physical coordinates to image coordinates and use either **invMapPoint()** or **inverse()** to map from image coordinates to physical coordinates.

Since the **cc2XformCalib2** is asymmetric, it is important to specify the positions in the **ccCrspPairVector** in the proper order. Furthermore, the **ccCrspPairVector** used to initialize a **cc2XformCalib2** transform should always be paired with the image coordinate point first, as shown in the figure above.

Constructors/Destructors

cc2XformCalib2

```
cc2XformCalib2();
cc2XformCalib2(const cc2XformCalib2& calibXform);
~cc2XformCalib2();
```

- `cc2XformCalib2();`

  Constructs an identity transform. The direction of the transform is set to map points from physical space to client space (usually the same as image space).

- `cc2XformCalib2(const cc2XformCalib2& calibXform);`

  Copy constructor.

  Parameters

  calibXform The new object that is a copy of this object.

- `~cc2XformCalib2();`

  Destructor.

Operators

operator=

```
cc2XformCalib2& operator=(
  const cc2XformCalib2& calibXform);
```

Assignment operator. Make this object a copy of *calibXform*.

Parameters

calibXform Source of the assignment.

operator*

```
using cc2XformBase::operator*;
```

Uses the same operator* as the base class.

Public Member Functions

init

```
void init(
  const ccCalib2ParamsIntrinsic& intrinsicParams,
  const ccCalib2ParamsExtrinsic& extrinsicParams);

void init(const ccCalib2CrspVector& correspondences);

void init(
  const ccCalib2CrspVector& correspondences,
  cmStd vector<double>& rmsResiduals,
  cmStd vector<double>& maxResiduals);
```

You initialize the object with intrinsic and extrinsic parameters, or with corresponding image points and physical points. The most recent initialization reflects the active transform. The direction of the transform is set to map points from physical space to client space (usually the same as image space).

- ```
  void init(
    const ccCalib2ParamsIntrinsic& intrinsicParams,
    const ccCalib2ParamsExtrinsic& extrinsicParams);
  ```

  Creates a calibration transform using the provided intrinsic and extrinsic parameters.

  Parameters

  intrinsicParams The intrinsic parameter set.

  extrinsicParams The extrinsic parameter set.

- ```
  void init(const ccCalib2CrspVector& correspondences);
  ```

  Create a calibration transform computed from a vector of corresponding image points and physical points. The direction of the transform is from the second coordinate space in the correspondences to the first coordinate space. If using results returned by the feature extractor provided with CVL (**cfCalib2VertexFeatureExtract()**), the direction of transformation is from physical space to client space (usually the same as image space).

The extrinsic parameters within this transform are set according to the first correspondence in the **ccCalib2CrspVector** input vector (index 0).

The number of views equals the size of input vector, one entry per view. The correspondences are generated from different views of a planar object. The following conditions must be met:

1. All intrinsic camera settings are expected to stay constant over all the views.
2. Two views are considered different if the planes of the objects in two views are not parallel (relative to the camera).

Parameters

correspondences
A vector of corresponding image points and physical points.

Throws

cc2XformCalibDefs::InsufficientPoints
If the number of points in any *ccCrspPairVector* is < 9.

*cc2XformCalibDefs::*NumerircalError
If computation cannot be further carried out. This may happen if conditions above are not met.

*cc2XformCalibDefs::*BadParams
If size of the **ccCalib2CrspVector** vector is 0.

Notes

If the number of views = 1, separation of intrinsic and extrinsic parameters is not possible.

As the number of views increases, the accuracy of computed intrinsic and extrinsic parameters increases.

The **cc2XformCalib2** is not symmetric between image coordinates and physical coordinates. The **cc2XformCalib2** transform should map from physical coordinates to image coordinates. The first cc2Vect in each ccCrspPair should represent the image location of the vertex expressed in image coordinates, and the second cc2Vect should represent the corresponding location in physical coordinates.

- ```
  void init(
    const ccCalib2CrspVector& correspondences,
    cmStd vector<double>& rmsResiduals,
    cmStd vector<double>& maxResiduals);
  ```

  Create a calibration transform computed from a vector of correspondences. All comments following the previous **init()** overload above are valid here too. Additionally, compute the residual errors in mapping points. The vectors *rmsResiduals* and *maxResiduals* are resized to the size of the *correspondences* vector.

  For single view calibration, rms and maximum residual errors of mapped points from "to" points in the single correspondence provided are computed and stored as single elements of *rmsResiduals* and *maxResiduals* respectively.

  For multiple view calibration, residual errors for the i-th correspondence are computed as follows:

  1. A **cc2XformCalib2** object T_i is constructed using intrinsic parameters of this **cc2XformCalib2** object and extrinsic parameters computed from the i-th correspondence.
  2. "from" points in the i-th correspondence are mapped through T_i.
  3. Rms and maximum residual errors of mapped points from "to" points in the i-th correspondence are computed and stored as the i-th element of *rmsResiduals* and *maxResiduals* respectively.

  Notes

  The **cc2XformCalib2** is not symmetric between image coordinates and physical coordinates. The **cc2XformCalib2** transform should map from physical coordinates to image coordinates. The first cc2Vect in each ccCrspPair should represent the image location of the vertex expressed in image coordinates, and the second cc2Vect should represent the corresponding location in physical coordinates.

intrinsicParams

```
const ccCalib2ParamsIntrinsic& intrinsicParams() const;
```

Returns the intrinsic calibration parameters which are part of this calibration transform.

Throws

cc2XformCalibDefs::BadParams

If this transform was initialized with a single view.
```

extrinsicParams

```
const ccCalib2ParamsExtrinsic& extrinsicParams() const;

void extrinsicParams(
  const ccCalib2ParamsExtrinsic& extrinsicParams);

ccCalib2ParamsExtrinsic extrinsicParams (
  const ccCrspPairVector& corr) const;

void extrinsicParams(const ccCrspPairVector& corr);
```

- `const ccCalib2ParamsExtrinsic& extrinsicParams() const;`

  Returns the extrinsic calibration parameters that are part of this calibration transform.

  Throws

  cc2XformCalibDefs::BadParams
  If this transform was initialized with a single view.

- `void extrinsicParams(`
  `  const ccCalib2ParamsExtrinsic& extrinsicParams);`

  Sets new extrinsic calibration parameters.

  Parameters

  extrinsicParams The new extrinsic calibration parameters.

  Throws

  cc2XformCalibDefs::BadParams
  If this transform was initialized with a single view.

- `ccCalib2ParamsExtrinsic extrinsicParams (`
  `  const ccCrspPairVector& corr) const;`

  Returns the extrinsic parameters associated with the view specified by *corr*.

  Notes

  The intrinsic camera calibration parameters used to obtain the provided correspondence must be same parameters that are part of this transform. In other words, the intrinsic camera settings used when **init()** was called must be unchanged when this function is called.

  Parameters

  corr The corresponding image point - physical point vector for one view.

Throws

cc2XformCalibDefs::BadParams
If this transform was initialized with a single view.

- `void extrinsicParams(const ccCrspPairVector& corr);`

Sets the extrinsic parameters associated with the view specified by *corr*.

Notes

The intrinsic camera calibration parameters used to obtain the provided correspondence must be same parameters that are part of this transform. In other words, the intrinsic camera settings used when **init()** was called must be unchanged when this function is called.

Calling **init()** with multiple views sets the extrinsic parameters in the transform to extrinsic parameters for the first view (index 0). If you wish the change the transform to use extrinsic parameters for a different view, you call this function.

Parameters

corr The corresponding image point - physical point vector for one view.

Throws

cc2XformCalibDefs::BadParams
If this transform was initialized with a single view.

extrinsicXform

```
cc3Xform extrinsicXform() const;

void extrinsicXform(const cc3Xform& rigidXform);

cc3Xform extrinsicXform(
  const ccCrspPairVector& corr) const;
```

- `cc3Xform extrinsicXform() const;`

Returns the rigid 3D transform between the physical 3D coordinate system and camera 3D coordinate system that is part this **cc2XformCalib2** transform. The direction of returned transform is same as direction of this **cc2XformCalib2** transform.

- `void extrinsicXform(const cc3Xform& rigidXform);`

Sets the rigid 3D transform between the camera coordinate system and the physical coordinate system to provided a rigid 3D transform. If matrix R_{rot} in *rigidXform* is not exactly a rotation matrix, it attempts to compute a rotation matrix closest to R_{rot} and then sets extrinsic transform. Such a computation should succeed in practical situations.

Parameters

rigidXform The new 3D transform.

Throws

cc2XformCalib2Defs::BadParams
If computation of a rotation matrix (if necessary) does not succeed.

- ```
  cc3Xform extrinsicXform(
    const ccCrspPairVector& corr) const;
  ```

Get the 3D rigid transform between the physical 3D coordinate system and camera 3D coordinate system as specified by the correspondence *coor*. The direction of returned transform is same as direction of this **cc2XformCalib2** transform.

Parameters

corr The correspondence vector.

mapPoint

```
virtual cc2Vect mapPoint(const cc2Vect& pt) const;
```

Maps a given 2-D point through this transform and returns the mapped point.

Parameters

pt The point to be mapped.

Throws

cc2XformCalibDefs::MapsToInfity
If *pt* maps to infinity through this transform.

isLinear

```
virtual bool isLinear() const;
```

Returns true if this transform is an affine transform (a linear transform). For example, the transform is linear if there is no perspective or radial distortion.

Notes

This function is designed to efficiently identify cases where transformations are guaranteed to be linear. The function may return false if it would take too much time to determine if it is linear.

linearXform

```
virtual cc2XformLinear linearXform(
  const cc2Vect& atThisPt) const;
```

Returns a linear transform which is the best approximation to this transform at the point *atThisPt*.
```

Parameters

atThisPt The point where the transform is linearized.

composeBase

```
virtual cc2XformBasePtrh composeBase(
  const cc2XformBase& rhs) const;
```

Returns the transform which is the composition of this transform and the transform *rhs*. Mapping a point with the resulting transformation is equivalent to first mapping the point by the transformation *rhs* and then by this transformation.

Parameters

rhs The transform to compose with this transform.

Notes

The only allowed compositions are: **cc2XformLinear** ***cc2XformLinear**, **cc2XformLinear** ***cc2XformPoly**, **cc2XformPoly** ***cc2XformLinear**

inverseBase

```
virtual cc2XformBasePtrh inverseBase() const;
```

Returns the inverse of this transform.

Notes

In the case of nonlinear transformations, the inverse transformation may not be exact.

clone

```
virtual cc2XformBasePtrh clone() const;
```

Returns a newly allocated copy of this object.

Notes

This function is required for polymorphic copying.

distortionModel

```
cc2XformCalib2Defs::ccCalib2DistortionModel
   distortionModel() const ;
```

Returns a distortion model used by this object.

Typedefs

ccCalib2CrspVector

```
typedef cmStd vector<ccCrspPairVector> ccCalib2CrspVector;
```

ccCrspPair

```
typedef ccPair<cc2Vect> ccCrspPair;
```

ccCrspPairVector

```
typedef cmStd vector<ccCrspPair> ccCrspPairVector;
```

cc2XformCalib2

cc2XformDeform

```
#include <ch_cvl/xfdeform.h>

class cc2XformDeform : public cc2XformBase
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class models a 2D coordinate deformation transform.

Constructors/Destructors

cc2XformDeform

```
cc2XformDeform();

cc2XformDeform(
  const cmStd vector<cc2Vect>& toPoints,
  const cmStd vector<cc2Vect>& fromPoints,
  double smoothness = 0);

cc2XformDeform(const cc2XformDeform& rhs);

~cc2XformDeform();
```

- ```
  cc2XformDeform();
  ```

  Default constructor. Constructs an identity transform.

- ```
  cc2XformDeform(
    const cmStd vector<cc2Vect>& toPoints,
    const cmStd vector<cc2Vect>& fromPoints,
    double smoothness = 0);
  ```

  Constructs a transform that maps *fromPoints* to *toPoints* using a smoothing parameter.

Notes

A smoothness of 0 yields an exact mapping at the specified point while a smoothness of infinity yields an affine transform.

The transform is non singular whenever points in the *fromPoints* vector are pair-wise distinct and not colinear.

Parameters

toPoints A vector of *from* points.

fromPoints A vector of *to* points.

smoothness The smoothing parameter.

Throws

cc2XformDeformDefs::BadParams
If *smoothness* < 0,
or if **toPoints.size()** != **fromPoints.size()**,
or if **fromPoints.size()** < 3.

ccMathError::Singular
If the transform is singular.

- `cc2XformDeform(const cc2XformDeform& rhs);`

Copy constructor.

- `~cc2XformDeform();`

Destructor

Operators

operator= `cc2XformDeform& operator=(const cc2XformDeform& rhs);`

Assignment operator.

operator* `cc2XformBasePtrh operator*(const cc2XformBase& rhs) const;`

Returns the composition of this transformation with the transformation *rhs*. Mapping a point with the resulting transform is equivalent to first mapping the point by the transform *rhs* and then by this transform. The use of this operator is equivalent of **composeBase(***rhs***)**.

Parameters

rhs The transform to be composed with this transform.

Public Member Functions

update

```
void update(
  const cmStd vector<cc2Vect>& toPoints,
  const cmStd vector<c_Bool>& enabled =
                                    cmStd vector<c_Bool>());
```

Updates this transform using the provided set of *toPoints*. *toPoint-fromPoint* correspondence may be enabled or disabled through the *enabled* vector as follows:

If the i-th element in *enabled* is ceTrue, then the i-th point in *fromPoints* (provided through the constructor) is mapped to the i-th point in *toPoints*. Otherwise, the i-th points in the *fromPoints* and *toPoints* vectors are both ignored by this transform.

Parameters

toPoints	The new to points vector.
enabled	The enable/disable vector.

Throws

cc2XformDeformDefs::BadParams
If **toPoints.size()** != **fromPoints.size()**,
or if **enabled.size()** != **fromPoints.size()**,
or if the number of ceTrue elements in *enabled* is < 3.

smoothness

```
double smoothness() const;
```

Returns the current smoothness value.

mapPoint

```
virtual cc2Vect mapPoint(const cc2Vect& pt) const;
```

Returns the point *pt*, mapped by this deformation transform.

This function is an override to the base class **cc2XformBase**.

Parameters

pt	The point to map.

isLinear

```
virtual bool isLinear() const;
```

Returns true if this deformation transform is linear. Returns false otherwise.

This function is an override to the base class **cc2XformBase**.

cc2XformDeform

linearXform

```
virtual cc2XformLinear linearXform(
  const cc2Vect& atThisPt) const;
```

Returns a linear transform which represents the best linearization of this transform at the given point. This linearization will map the given point exactly but will likely be accurate only in a small region around that point.

This function is an override to the base class **cc2XformBase**.

Parameters

atThisPt The point where the deformation transform is to be linearized.

inverseBase

```
virtual cc2XformBasePtrh inverseBase() const;
```

Returns a transform which is the inverse of this transform.

This function is an override to the base class **cc2XformBase**.

Notes

The inverse will not be exact unless this deformation transform is linear.

This function will take time comparable to the time taken to construct the transform to be inverted. It is not a fast operation.

Throws

ccMathError::Singular
If this transform cannot be inverted because of singularity.

clone

```
virtual cc2XformBasePtrh clone() const;
```

Returns a newly allocated copy of this object.

This function is an override to the base class **cc2XformBase**.

Notes

This function is required for polymorphic copying.

isIdentity

```
bool isIdentity() const;
```

Returns true if this transform is an identity transform.

This function is an override to the base class **cc2XformBase**.

cc2XformLinear

```
#include <ch_cvl/xflinear.h>

class cc2XformLinear : public cc2XformBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class implements a linear transformation between coordinate systems. If (C_x, C_y) are the client coordinates of generic image features and (I_x, I_y) their corresponding image coordinates, the mapping performed by the class is:

$$\begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} I_x \\ I_y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

where:

$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ is the matrix component of the linear transformation

$\begin{bmatrix} T_x \\ T_y \end{bmatrix}$ is the translation vector component of the linear transformation

A generic linear transformation L will be referred to as the pair:

$$L = \left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \begin{bmatrix} T_x \\ T_y \end{bmatrix} \right)$$

Notes

If you use this class to map image features from frame A to frame B, the components of the translation vector must be the ones of the vector that goes from frame B to frame A expressed in Frame B (see *Math Foundations of Transformations* in the *CVL User's Guide)*. If the translation vector is not specified in this way, the results returned by the mapping functions will not be correct.

Constructors/Destructors

cc2XformLinear

```
cc2XformLinear();

cc2XformLinear(const cc2Matrix& c, const cc2Vect& t);

cc2XformLinear(const cc2Vect& t, const ccRadian& xRot,
  const ccRadian& yRot, double xScale, double yScale);

cc2XformLinear(const cc2Vect& t, double scale,
  double aspect, const ccRadian& shear,
  const ccRadian& rotation);

cc2XformLinear(const cc2Xform&);
```

- `cc2XformLinear();`

  The default constructor initializes this transformation to the identity transformation.The identity transformation maps a point to itself.

  Identity Transformation: $\left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \begin{bmatrix}0\\ 0\end{bmatrix}\right)$

- `cc2XformLinear(const cc2Matrix& c, const cc2Vect& t);`

  Initializes this transformation to the matrix *c* and the translation vector *t*.

  Parameters

c	The matrix this transformation is set to
t	The translation vector this transformation is set to

- `cc2XformLinear(const cc2Vect& t, const ccRadian& xRot, const ccRadian& yRot, double xScale, double yScale);`

  Initializes the matrix component of this transformation according to the scale-rotation mode (see *Math Foundations of Transformations* and *Images and Coordinates* in the *CVL User's Guide*). The translation vector component of this transformation is set to *t*.

  Parameters

t	The translation vector this transformation is set to.
xRot	The x-rotation; the amount by which to rotate the x-axis in radiants.
yRot	The y-rotation; the amount by which to rotate the y-axis in radiants.
xScale	The x-scale; the amount by which to scale the x-axis units.
yScale	The y-scale; the amount by which to scale the y-axis units.

- `cc2XformLinear(const cc2Vect& t, double scale, double aspect, const ccRadian& shear, const ccRadian& rotation);`

  Initializes the matrix component of this transformation according to the shear-aspect mode (see *Math Foundations of Transformations* and *Images and Coordinates* in the *CVL User's Guide*). The translation vector component of this transformation is set to *t*.

  Parameters

t	The translation vector this transformation is set to.
scale	The scale element; the amount by which to scale all units.
aspect	The aspect ratio; the ratio of the x-axis units to the y-axis units.
shear	The shear element; the amount by which to rotate the y-axis in radiants.
rotation	The rotation element; the amount by which to rotate the entire coordinate system in radiants.

- `cc2XformLinear(const cc2Xform& c);`

  Initializes this transformation to the **cc2Xform** transformation *c*.

  Parameters

c	The **cc2Xform** transformation this transformation is set to

Notes

This constructor is primarily for convenience when using existing CVL code.

Operators

cc2Xform

```
operator cc2Xform() const;
```

Cast this transformation to its equivalent **cc2Xform** form.

operator==

```
bool operator== (const cc2XformLinear& c) const;
```

Returns true if all the components of this transformation are the same as the corresponding components of *m*.

Parameters

c	The transformation compared to this transformation

operator!=

```
bool operator!= (const cc2XformLinear& c) const;
```

Returns true if this transformation is not the same as *c*.

Parameters

c	The transformation compared to this transformation

operator=

```
cc2XformLinear& operator=(const cc2Xform& rhs);
```

Assignment operator. Creates a new **cc2XformLinear** object equal to *rhs*.

Parameters

rhs	The assignment source.

operator*

```
cc2XformLinear operator*(
  const cc2XformLinear& xform) const;

cc2XformBasePtrh operator*(const cc2XformBase& base) const;

cc2Vect operator*(const cc2Vect& pt) const;
```

- ```
  cc2XformLinear operator*(
    const cc2XformLinear& xform) const;
  ```

  Returns the composition of this transform with the transform *xform*. Mapping a point with the resulting transform is equivalent to first mapping the point by the transform *xform* and then by this transform. The use of this operator is equivalent to the use of the **compose()** function.
  ```

Parameters

xform The transform to be composed with this transform

- `cc2XformBasePtrh operator*(const cc2XformBase& base) const;`

Returns the composition of this transform with the base transform *base*. Mapping a point with the resulting transform is equivalent to first mapping the point by the transform *base* and then by this transform. The use of this operator is equivalent to the use of the **composeBase()** function.

Parameters

base The base transform.

- `cc2Vect operator*(const cc2Vect& pt) const;`

Maps the point *pt* by using this transform. This is equivalent to calling **mapPoint()** (see below).

Parameters

pt The point to map.

Public Member Functions

mapPoint `virtual cc2Vect mapPoint(const cc2Vect& pt) const;`

Maps the point *pt* by using this transformation (see *Math Foundations of Transformations* in the *CVL User's Guide*).

If $\left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \begin{bmatrix} T_x \\ T_y \end{bmatrix}\right)$ is this transformation and $\begin{bmatrix} pt_x \\ pt_y \end{bmatrix}$ is the point *pt,* the point returned by the function is::

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} pt_x \\ pt_y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

Parameters

pt The point to map.

isLinear `virtual bool isLinear() const;`

Always returns true since this transformation is linear.

cc2XformLinear

linearXform

```
virtual cc2XformLinear linearXform(const cc2Vect& atThisPt)
  const;
```

This function returns a transformation identical to this transformation regardless of *atThisPt*.

Parameters

atThisPt The point around which this transformation is linearized.

composeBase

```
virtual cc2XformBasePtrh composeBase
  (const cc2XformBase& rhs) const;
```

Returns a pointer handle to the transformation that is the composition of this transformation and *rhs*. This transformation can be composed with either a **cc2XformLinear** or a **cc2XformPoly** transformation.

Parameters

rhs The transformation to be composed with this transformation.

inverseBase

```
virtual cc2XformBasePtrh inverseBase() const;
```

Returns a pointer handle to the inverse of this transformation.

Throws

ccMathError::Singular
The transformation is singular.

clone

```
virtual cc2XformBasePtrh clone() const;
```

Returns a newly allocated copy of this transformation.

matrix

```
const cc2Matrix& matrix();

void matrix(const cc2Matrix& c);
```

- ```
  const cc2Matrix& matrix();
  ```

  Returns the matrix component of this linear transformation.

- ```
  void matrix(const ccMatrix<2>& c);
  ```

  Sets the matrix component of this linear transformation to *c*.

  Parameters

  c The matrix this transformation's matrix is set to.

trans

```
const cc2Vect& trans();
void trans(const cc2Vect& t);
```

- `const cc2Vect& trans();`

  Returns the 2-element translation vector of this transformation.

- `void trans(const cc2Vect& t);`

  Sets the translation vector component of this transformation to *t*.

  Parameters

  t — The vector this transformation's vector is set to.

inverse

```
cc2XformLinear inverse() const;
```

Returns the inverse of this transformation

Throws

ccMathError::Singular
The transformation is singular.

compose

```
cc2XformLinear compose(const cc2XformLinear& xform) const;
```

Returns the transformation resulting from the composition of this transformation with *xform*. Mapping a point with the resulting transformation is equivalent to first mapping the point by the transformation *xform* and then by this transformation.

Parameters

xform — The linear transformation to be composed with this transformation.

xRot

```
ccRadian xRot() const;
```

Returns the x-rotation element of this transformation when using the scale-rotation specification.

Throws

ccMathError::Singular
The transformation is singular.

yRot `ccRadian yRot() const;`

Returns the y-rotation element of this transformation when using the scale-rotation specification.

Throws

ccMathError::Singular
The transformation is singular

xScale `double xScale() const;`

Returns the x-scale element of this transformation when using the scale-rotation specification.

Throws

ccMathError::Singular
The transformation is singular.

yScale `double yScale() const;`

Returns the y-scale element of this transformation when using the scale-rotation specification.

ccMathError::Singular
The transformation is singular

scale `double scale() const;`

Returns the scale element of this transformation when using the shear-aspect specification.

Throws

ccMathError::Singular
The transformation is singular.

aspect `double aspect() const;`

Returns the aspect element of this transformation when using the shear-aspect specification.

Notes

Be careful when you mix elements from the scale-rotation specification with elements of the shear-aspect specification. For example, if you specify a shear angle in the shear-aspect specification, then **aspect()**!= **yScale() / xScale()**.

Throws

ccMathError::Singular
The transformation is singular.

shear `ccRadian shear();`

Returns the shear element of this transformation when using the shear-aspect specification.

Throws
ccMathError::Singular
The transformation is singular.

rotation `ccRadian rotation() const;`

Returns the rotation element of this transformation when using the shear-aspect specification.

Throws
ccMathError::Singular
The transformation is singular.

mapAngle `ccRadian mapAngle (const ccRadian& ang) const;`

Return the results of mapping the angle *ang* with this transformation.

Parameters
ang The angle to map in radians.

invMapAngle `ccRadian invMapAngle (const ccRadian& ang) const;`

Return the results of mapping the angle *ang* with the inverse of this transformation.

Parameters
ang The angle to map in radians.

Throws
ccMathError::Singular
The transformation is singular.

invMapPoint `cc2Vect invMapPoint (const cc2Vect &pt) const;`

Returns the result of mapping the point *pt* with the inverse of this transformation

If $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ and $\begin{bmatrix} T_x \\ T_y \end{bmatrix}$ are the matrix and translation vector components of this

transformation and $\begin{bmatrix} pt_x \\ pt_y \end{bmatrix}$ is the point *pt,* the point returned by the function is:

$$\frac{1}{det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \begin{bmatrix} pt_x - T_x \\ pt_y - T_y \end{bmatrix}$$

mapVector

```
cc2Vect mapVector (const cc2Vect &vect) const;
```

Maps the vector *vect* by using the matrix component of this transformation. If $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ is the matrix component of this transformation, the vector returned by the function is:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} vect_x \\ vect_y \end{bmatrix}$$

Parameters

vect The vector to be mapped.

invMapVector

```
cc2Vect invMapVector (const cc2Vect &vect) const;
```

Maps the vector *vect* by using the inverse of the matrix component of this transformation. If $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ is the matrix component of this transformation, the vector returned by the function is:

$$\frac{1}{det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \begin{bmatrix} vect_x \\ vect_y \end{bmatrix}$$

Parameters

vect The vector to be mapped.

Throws

ccMathError::Singular
The transformation is singular.

mapArea

```
double mapArea (double area) const;
```

Returns the area after mapping by this transformation.

Parameters

area The area to be mapped.

Notes

This function computes the mapped area by first calculating the area of a unit square and then multiplying area by that value. You may wish to call this function with the area of 1.0 to get a conversion constant you can use without the overhead of calling this function multiple times.

invMapArea

```
double invMapArea (double area) const;
```

Returns the area after mapping by the inverse of this transformation.

Parameters

area The area to be mapped.

Throws

ccMathError::Singular
The transformation is singular.

isIdentity

```
bool isIdentity() const;
```

Returns true if this transformation is the identity transformation (in this case all points map to themselves).

isSingular

```
bool isSingular() const;
```

Returns true if this transformation is singular (in this case all points map to the same point or line).

Data Members

I

```
static const cc2XformLinear I;
```

The identity transformation for **cc2XformLinear**.

cc2XformLinear

cc2XformPerspective

```
#include <ch_cvl/xfpersp.h>

class cc2XformPerspective: public cc2XformBase
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class models a perspective transform used for viewing a planar object. In general, this is a nonlinear transform represented by a 3x3 matrix, for example the matrix M.

The transform maps a 2D point $\begin{bmatrix} x \\ y \end{bmatrix}$ to the 2D point $\begin{bmatrix} u/w \\ v/w \end{bmatrix}$

Where $\begin{bmatrix} u \\ v \\ w \end{bmatrix} = M \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

The matrix *M* is a 3x3 matrix as follows:

$$M = \begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{bmatrix}$$

For more details, please see *Multiple View Geometry in Computer Vision* by Richard Hartley and Andrew Zisserman.

Constructors/Destructors

cc2XformPerspective

```
cc2XformPerspective();

cc2XformPerspective(
  const cmStd vector<cc2Vect>& toPoints,
  const cmStd vector<cc2Vect>& fromPoints);

cc2XformPerspective(
  const cc2Vect& cornerPo,
  const cc2Vect& cornerPx,
  const cc2Vect& cornerPy,
  const cc2Vect& cornerPopp);

cc2XformPerspective(const cc3Matrix& mat);

cc2XformPerspective(const cc2XformPerspective& xform);

~cc2XformPerspective();
```

- `cc2XformPerspective();`

  Constructs the identity transform.

- ```
  cc2XformPerspective(
    const cmStd vector<cc2Vect>& toPoints,
    const cmStd vector<cc2Vect>& fromPoints);
  ```

  Constructs a perspective transformation that achieves a least-squares best fit from *fromPoints* to *toPoints*.

  Parameters

  toPoints The transform to points.

  fromPoints The transform from points.

  Throws

  cc2XformPerspectiveDefs:: BadParams
  If **fromPoint.size()** != **toPoints.size()**, or if **fromPoints.size()** < 4.

  ccMathError::Singular
  If a transform cannot be determined. This can occur if the points all lie on a single point or line.
  ```

- ```
  cc2XformPerspective(
    const cc2Vect& cornerPo,
    const cc2Vect& cornerPx,
    const cc2Vect& cornerPy,
    const cc2Vect& cornerPopp);
  ```

  Constructs a perspective transform that maps:

  (0, 0) --> *cornerPo*
  (1, 0) --> *cornerPx*
  (0, 1) --> *cornerPy*
  (1, 1) --> *cornerPopp*

  Parameters

cornerPo	Location of the *Po* corner.
cornerPx	Location of the *Px* corner.
cornerPy	Location of the *Py* corner.
cornerPopp	Location of the *Popp* corner.

  Notes

  If the provided corners are set to identical points, the derived transform may be unable to map the corner of the unit square.

- ```
  cc2XformPerspective(const cc3Matrix& mat);
  ```

  Constructs a perspective transform with matrix *mat*. Nonzero scalar multiples of a matrix produce the same perspective transform.

  Parameters

mat	The supplied 3D matrix.

- ```
  cc2XformPerspective(const cc2XformPerspective& xform);
  ```

  Copy Constructor

- ```
  ~cc2XformPerspective();
  ```

  Destructor.

Operators

operator=

```
const cc2XformPerspective& operator=(
  const cc2XformPerspective& that);
```

Assignment operator. Make this object a copy of *that*.

Parameters

that The object to copy.

operator==

```
bool operator==(const cc2XformPerspective& that) const;
```

Returns true if each entry in the matrix of *this* is equal to each corresponding entry in the matrix of *that*. Returns false otherwise.

Parameters

that The transform to compare with this transform.

Public Member Functions

matrix

```
cc3Matrix matrix() const;
```

Returns the 3x3 matrix of this perspective transform.

mapPoint

```
virtual cc2Vect mapPoint(const cc2Vect& pt) const;
```

Returns the 2D point *pt mapped* through this transform.

Parameters

pt The point to map.

Throws

ccMathError::Singular
If computation involves division by zero.

isLinear

```
virtual bool isLinear() const;
```

Returns true if this transform is exactly linear; returns false otherwise.

Using the notation in the introduction, this transform is exactly linear if *m31* = *m32* = 0.

linearXform

```
virtual cc2XformLinear linearXform(const cc2Vect& pt)
                                                    const;
```

Returns a linear transform which represents the best approximation to this perspective transform at 2D point *pt*.

Parameters

pt The point where the linear approximation is made.

Throws

ccMathError::Singular
If computation involves division by zero.

composeBase

```
virtual cc2XformBasePtrh composeBase(
  const cc2XformBase& rhs) const;
```

Returns a transform which is the composition of *this* with the given transform. For example, (**this*) * *rhs*.

Parameters

rhs The transform to compose with this transform.

inverseBase

```
virtual cc2XformBasePtrh inverseBase() const;
```

Returns the inverse of this transform.

Throws

ccMathError::Singular
If this transform is singular.

clone

```
virtual cc2XformBasePtrh clone() const;
```

Returns a newly allocated copy of this object.

Notes

This function is required for polymorphic copying.

isSingular

```
bool isSingular() const;
```

Returns true if this transform is singular; returns false otherwise.

cc2XformPerspective

cc2XformPoly

```
#include <ch_cvl/xfpoly.h>

class cc2XformPoly : public cc2XformBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class implements a nonlinear polynomial mapping between coordinate frames to be used in all those instances where a high level of mapping accuracy is required. The class implements polynomial mappings of order 1, 3 and 5 (for more details, see *Math Foundations of Transformations* in the *CVL User's Guide*). If (C_x, C_y) are client coordinates of generic image features and (I_x, I_y) are their corresponding image coordinates, the mapping performed by the class is:

$$C_x = \sum_{i,j \geq 0}^{i+j \leq n} a_{ij} {I_x}^i {I_y}^j \qquad C_y = \sum_{i,j \geq 0}^{i+j \leq n} b_{ij} {I_x}^i {I_y}^j$$

where a_{ij} and b_{ij} are the coefficients of the transformation and n is the degree of the transformation.

Constructors/Destructors

cc2XformPoly

```
cc2XformPoly();

cc2XformPoly(const cmStd vector<cc2Vect>& toPoints,
  const cmStd vector<cc2Vect>& fromPoints,
  c_Int32 order = 5);
```

- `cc2XformPoly();`

  Constructs the identity transformation. The identity transformation is:

  $C_x = I_x \qquad C_y = I_y$

- ```
  cc2XformPoly(const cmStd vector<cc2Vect>& toPoints,
    const cmStd vector<cc2Vect>& fromPoints,
    c_Int32 order = 5);
  ```

  Creates a polynomial transformation of order *order*. The coefficients of the transformation computed by the constructor provide the best least-square fit between the list of points *fromPoints* (typically in image coordinates) and the list of points *toPoints* (typically in client coordinates). Only polynomials of orders 1, 3 and 5 are allowed. The following table lists the minimum number of points for each order

Order	Number of Points
1	3
3	10
5	21

Parameters

toPoints The target list of points in client coordinates.

fromPoints The starting list of points in image coordinates.

order The order of the polynomial transformation. It must be 1, 3 or 5.

Notes

In general, the more points are used the more accurate the resulting polynomial fit will be. It is recommended that a grid of at least 8x8 points, spanning the entire region of interest, is used.

Throws

cc2XformPoly::BadParams

Not enough points in *fromPoints* and *toPoints* for the requested order of the polynomial transformation.

The number of points in *fromPoints* is not equal to the number of points in *toPoint*s.

The requested order of the polynomial transformation is less than 1.

cc2XformDefs::NotImplemented

The requested order of the polynomial is positive and not 1, 3, or 5.

cc2XformPoly::IllDefined

The least-square fit cannot converge to a solution.
  ```

Public Member Functions

mapPoint

```
virtual cc2Vect mapPoint(const cc2Vect& fromPoint) const;
```

Maps the coordinates of the point *fromPoint* using this transformation.

Parameters

fromPoint The point mapped by this transformation

isLinear

```
virtual bool isLinear() const;
```

Returns true if the order of this transformation is 1 or if all the coefficients of the polynomial terms with degree greater than one are exactly zero.

linearXform

```
virtual cc2XformLinear linearXform(const cc2Vect& atThisPt)
  const;
```

Returns the linear transformation defined by the linear terms of the Taylor series expansion of this transformation about the point *atThisPt* (for more details see *Math Foundations of Transformations* in the *CVL User's Guide*).

Parameters

atThisPt The point where the returned **cc2XformLinear** transformation is defined

composeBase

```
virtual cc2XformBasePtrh composeBase(
  const cc2XformBase& rhs)const;
```

Returns a transformation that is the composition of this transformation with *rhs*. Mapping a point with the resulting transformation is equivalent to first mapping the point by *rhs* and then by this transformation. This transformation can only be composed with a **cc2XformLinear** object.

Parameters

rhs The transformation composed with this transformation

Throws

cc2XformDefs::NotImplemented
This transformation is composed with an object other than **cc2XformLinear**.

inverseBase

```
virtual cc2XformBasePtrh inverseBase() const;
```

Returns the inverse of this transformation. The inverse of this transformation maps points in client coordinates to points in image coordinates.

Notes

This function may not provide an exact inverse mapping.

Throws

ccMathError::Singular

The transformation is singular and cannot be inverted.

clone

```
virtual cc2XformBasePtrh clone() const;
```

Returns a newly allocated copy of this object.

cc3AngleVect

```
#include <ch_cvl/ccalib.h>

class cc3AngleVect;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class represents a rotation in 3-D space. It holds three angles which express rotations of the axes of a 3-D coordinate system. It is used to express the orientation of one coordinate system with respect to another. We use the convention of *Givens* rotations (see *Multiple View Geometry in Computer Vision*, Appendix A3.1.1, by *Hartley & Zisserman*).

A rotation R of 3-D coordinate axes is expressed as:

R = Rx * Ry * Rz

where

Rz = rotation of x,y-axes about a fixed z-axis,
Ry = rotation of z,x-axes about a fixed y-axis,
Rx = rotation of y,z-axes about a fixed x-axis.

The order of applying rotations is: Rz first, Ry second, and Rx third.

Constructors/Destructors

cc3AngleVect

```
cc3AngleVect(
  const ccRadian& angleX,
  const ccRadian& angleY,
  const ccRadian& angleZ);

cc3AngleVect();
```

- ```
  cc3AngleVect(
    const ccRadian& angleX,
    const ccRadian& angleY,
    const ccRadian& angleZ);
  ```

  Constructs an object with angles initialized to the provided values.
  ```

Parameters

angleX	The y,z-axes rotation angle about a fixed x-axis.
angleY	The z,x-axes rotation angle about a fixed y-axis.
angleZ	The x,y-axes rotation angle about a fixed z-axis.

- `cc3AngleVect();`

  Constructs an object with all three angles initialized to 0.

Public Member Functions

x

```
ccRadian x();
void x(const ccRadian& angleX);
```

- `ccRadian x();`

  Returns the y,z-axes rotation angle about a fixed x-axis.

- `void x(const ccRadian& angleX);`

  Sets a new y,z-axes rotation angle about a fixed x-axis.

  Parameters

angleX	The new y,z-axes rotation angle about a fixed x-axis.

y

```
ccRadian y() const;
void y(const ccRadian& angleY);
```

- `ccRadian y() const;`

  Returns the z,x-axes rotation angle about a fixed y-axis.

- `void y(const ccRadian& angleY);`

  Sets a new z,x-axes rotation angle about a fixed y-axis.

  Parameters

angleY	The new z,x-axes rotation angle about a fixed y-axis.

z

```
ccRadian z() const;
void z(const ccRadian& angleZ);
```

- `ccRadian z() const;`

  Returns the x,y-axes rotation angle about a fixed z-axis.

- `void z(const ccRadian& angleZ);`

  Sets a new x,y-axes rotation angle about a fixed z-axis.

  Parameters

  angleZ The new x,y-axes rotation angle about a fixed z-axis.

matrix

```
cc3Matrix matrix() const;
void matrix(const cc3Matrix& R);
```

- `cc3Matrix matrix() const;`

  Returns the rotation matrix associated with rotation of the 3-D coordinate axes through angles in this object. See the introduction to this reference page for the rotation order applied.

- `void matrix(const cc3Matrix& R);`

  Set the angles in this object to angles computed from rotation matrix *R*. Theoretically, *R* must satisfy determinant (*R*) = 1 and *R** (Transpose of *R*) = Identity. If R is not exactly a rotation matrix it attempts to compute a rotation matrix closest to *R* and then sets the angles. Such a computation should succeed in practical situations.

  Parameters

  R The new rotation matrix.

  Throws

  cc2XformCalib2Defs::BadParams
  If computation of a rotation matrix (if necessary) does not succeed.

cc3AngleVect

cc3PlanePel

```
#include <ch_cvl/colorpel.h>

class cc3PlanePel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **cc3PlanePel** class describes a three-plane pixel that can be used to create multi-planar pel buffers. The planes of the three-plane pixel are named plane 0, plane 1, and plane 2. The way the three planes are used depends on the application. One typical use is RGB, although HLS and HSV can also be supported.

In addition to this three-plane pixel class, CVL also defines additional pointer, **const** pointer, reference, and **const** reference classes that provide transparent pointer and reference semantics:

- **cc3PlanePtr**
- **cc3PlanePtr_const**
- **cc3PlaneRef**
- **cc3PlaneRef_const**

Constructors/Destructors

cc3PlanePel

```
cc3PlanePel();
cc3PlanePel(c_UInt8 ch1, c_UInt8 ch2, c_UInt8 ch3);
```

- `cc3PlanePel();`

  The default constructor creates a pixel with uninitialized elements.

- `cc3PlanePel(c_UInt8 ch1, c_UInt8 ch2, c_UInt8 ch3);`

  Creates a pixel with the specified values for each plane.

  Parameters

  ch1 Value for plane 0.

ch2	Value for plane 1.
ch3	Value for plane 2.

Public Member Functions

plane0

```
c_UInt8& plane0();
const c_UInt8& plane0() const;
```

- `c_UInt8& plane0();`

  Returns a read-write reference to the plane 0 element of the pixel.

- `const c_UInt8& plane0() const;`

  Returns a read-only reference to the plane 0 element of the pixel.

plane1

```
c_UInt8& plane1();
const c_UInt8& plane1() const;
```

- `c_UInt8& plane1();`

  Returns a read-write reference to the plane 1 element of the pixel.

- `const c_UInt8& plane1() const;`

  Returns a read-only reference to the plane 1 element of the pixel.

plane2

```
c_UInt8& plane2();
const c_UInt8& plane2() const;
```

- `c_UInt8& plane2();`

  Returns a read-write reference to the plane 2 element of the pixel.

- `const c_UInt8& plane2() const;`

  Returns a read-only reference to the plane 2 element of the pixel.

cc3PlanePelBuffer

```
#include <ch_cvl/colorbuf.h>

class cc3PlanePelBuffer : public ccPelBuffer<cc3PlanePel>;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **cc3PlanePelBuffer** class is an instance of the template class **ccPelBuffer** that uses three-plane pixels (see **cc3PlanePel** on page 145).

Constructors/Destructors

cc3PlanePelBuffer

```
cc3PlanePelBuffer();

cc3PlanePelBuffer(c_Int32 width, c_Int32 height,
  c_Int32 alignModulus=32);

cc3PlanePelBuffer(ccPelRoot<cc3PlanePel>* pelroot);
```

- `cc3PlanePelBuffer();`

  Creates an unbound window, that is, one that is not associated with any root image. Height and width are set to 0 and all offsets are set to (0,0). The window's transform is set to identity.

- `cc3PlanePelBuffer(c_Int32 width, c_Int32 height, c_Int32 alignModulus=32);`

  Allocates contiguous storage for a root image of *width* x *height* pixels of type *P* and a window that encompasses the entire root image. The pixels are default-constructed. This storage will be freed by the destructor. The byte address of the first row's first pixel is zero modulo the *alignModulus*. The allocated row width (in pixels) is increased if necessary to make the row size (in bytes) a multiple of the specified *alignModulus*. Up to (*alignModulus*-1) pad pixels may be allocated for each row. Pad pixels, if any, occur to the right (greater x) of the pixels that belong to the root image. Pixel processing routines are permitted to overwrite pad pixels.

Parameters

width	The width, in pixels, of the root image. (*width* > 0)
height	The height, in rows, of the root image. (*height* > 0)
alignModulus	A value that determines how memory used for the image is aligned. A value of 1 means that memory is aligned on byte boundaries, 2 aligns on word boundaries, 4 aligns on long word boundaries, and so on. (*alignModulus* >= 1)

- `cc3PlanePelBuffer(ccPelRoot<cc3PlanePel>* pelroot);`

Creates a window bound to the root image *pelroot*. Height and width are set to encompass the entire root image. Offset is set to (0,0) and the transform is set to identity.

Parameters

pelroot	The root image the window is bound to.

Notes

If *pelroot* is *NULL*, the effect is the same as **cc3PlanePelBuffer()**; this window becomes unbound.

Public Member Functions

plane0

```
ccPelBuffer_const<c_UInt8> plane0() const;
ccPelBuffer<c_UInt8> plane0();
```

- `ccPelBuffer_const<c_UInt8> plane0() const;`

Returns plane 0 of this pel buffer as a read-only grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

- `ccPelBuffer<c_UInt8> plane0();`

Returns plane 0 of this pel buffer as a read-write grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

plane1

```
ccPelBuffer_const<c_UInt8> plane1() const;
ccPelBuffer<c_UInt8> plane1();
```

- `ccPelBuffer_const<c_UInt8> plane1() const;`

  Returns plane 1 of this pel buffer as a read-only grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

- `ccPelBuffer<c_UInt8> plane1();`

  Returns plane 1 of this pel buffer as a read-write grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

plane2

```
ccPelBuffer_const<c_UInt8> plane2() const;
ccPelBuffer<c_UInt8> plane2();
```

- `ccPelBuffer_const<c_UInt8> plane2() const;`

  Returns plane 2 of this pel buffer as a read-only grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

- `ccPelBuffer<c_UInt8> plane2();`

  Returns plane 2 of this pel buffer as a read-write grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

colorSpace

```
ccColorSpaceDefs::ColorSpace colorSpace() const;
void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);
```

- `ccColorSpaceDefs::ColorSpace colorSpace() const;`

  Returns the color space (RGB or HSI) of the image.

- `void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);`

  Sets the color space (RGB or HSI) of the image.

  Parameters

colorSpace	The color space: **ccColorSpaceDefs::eRGB** or **ccColorSpaceDefs::eHSI**

cc3PlanePelBuffer_const

```
#include <ch_cvl/colorbuf.h>

class cc3PlanePelBuffer_const : public
                          ccPelBuffer_const<cc3PlanePel>
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **cc3PlanePelBuffer** class is an instance of the template class **ccPelBuffer** on page 2371 that uses three-plane pixels (see **cc3PlanePel** on page 145). This class allows read-only access to the planes of the pel buffer.

Constructors/Destructors

cc3PlanePelBuffer_const

```
cc3PlanePelBuffer_const();

cc3PlanePelBuffer_const(const ccPelRoot<cc3PlanePel>*);
```

- `cc3PlanePelBuffer_const();`

  Creates an unbound window, that is, one that is not associated with any root image. Height and width are set to 0 and all offsets are set to (0,0). The window's transform is set to identity.

- ```
  cc3PlanePelBuffer_const(
    const ccPelRoot<cc3PlanePel>* pelroot);
  ```

  Creates a window bound to the root image *pelroot*. Height and width are set to encompass the entire root image. Offset is set to (0,0) and the transform is set to identity.

  Parameters

  pelroot The root image the window is bound to.

  Notes

  If *pelroot* is *NULL*, the effect is the same as **cc3PlanePelBuffer_const()**; this window becomes unbound.
  ```

Public Member Functions

plane0

```
ccPelBuffer_const<c_UInt8> plane0() const;
```

Returns plane 0 of this pel buffer as a read-only grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

plane1

```
ccPelBuffer_const<c_UInt8> plane1() const;
```

Returns plane 1 of this pel buffer as a read-only grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

plane2

```
ccPelBuffer_const<c_UInt8> plane2() const;
```

Returns plane 2 of this pel buffer as a read-only grey-scale pel buffer. The returned pel buffer has the same window size, offset, root offset, and transform as this pel buffer. The returned pel buffer exists even if this pel buffer is destroyed.

colorSpace

```
ccColorSpaceDefs::ColorSpace colorSpace() const;
void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);
```

- `ccColorSpaceDefs::ColorSpace colorSpace() const;`

  Returns the color space (RGB or HSI) of the image.

- `void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);`

  Sets the color space (RGB or HSI) of the image.

  Parameters

colorSpace	The color space: **ccColorSpaceDefs::eRGB** or **ccColorSpaceDefs::eHSI**

cc3PlanePelPtr

```
#include <ch_cvl/colorpel.h>

class cc3PlanePelPtr : public cc3PlanePelPtr_const;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **cc3PlanePelPtr** class provides pointer semantics for the three-plane pixel class **cc3PlanePel** (see page 145). It is used by the **ccPelTraits** class (see page 2403).

Constructors/Destructors

cc3PlanePelPtr

```
cc3PlanePelPtr();
cc3PlanePelPtr(c_UInt8 *ch1, c_UInt8 *ch2, c_UInt8 *ch3);
explicit cc3PlanePelPtr(cc3PlanePel *pelPtr);
explicit cc3PlanePelPtr(const cc3PlanePelPtr_const &cPtr);
```

- `cc3PlanePelPtr();`

  Creates a three-plane pixel pointer that does not point to any pixel.

- `cc3PlanePelPtr(c_UInt8 *ch1, c_UInt8 *ch2, c_UInt8 *ch3);`

  Creates a three-plane pixel pointer that points to the three values supplied.

  Parameters

ch1	A pointer to the plane 0 value.
ch2	A pointer to the plane 1 value
ch3	A pointer to the plane 2 value.

- `explicit cc3PlanePelPtr(cc3PlanePel *pelPtr);`

  Creates a three-plane pixel pointer that points to the same three-plane pixel pointed to by *pelPtr*.

Parameters

pelPtr A pointer to a three-plane pixel object.

- `explicit cc3PlanePelPtr(const cc3PlanePelPtr_const &cPtr);`

  Converts a **cc3PlanePelPtr_const** object to **cc3PlanePelPtr** object.

  Parameters

  cPtr The **const** pointer to convert.

Public Member Functions

pPlane0 `c_UInt8* pPlane0() const;`

Returns a pointer to the plane 0 element of the pixel.

pPlane1 `c_UInt8* pPlane1() const;`

Returns a pointer to the plane 1element of the pixel.

pPlane2 `c_UInt8* pPlane2() const;`

Returns a pointer to the plane 2 element of the pixel.

Operators

These operators allow pointer arithmetic on pointers to three-plane pixels. As for pointers to scalars, the pixels are expected to be contiguous.

operator+ `cc3PlanePelPtr operator+(int d) const;`

Returns a pointer to the *d*th pixel beyond this one.

Parameters

d The distance from this pixel.

operator+= `cc3PlanePelPtr& operator+=(int d);`

Returns a reference the *d*th pixel beyond this one.

Parameters

d The distance from this pixel.

operator-

```
cc3PlanePelPtr operator-(int d) const;
```

Returns a pointer to the *d*th pixel before this one.

Parameters

d	The distance from this pixel.

operator-=

```
cc3PlanePelPtr& operator-=(int d);
```

Returns a reference the *d*th pixel before this one.

Parameters

d	The distance from this pixel.

operator++

```
cc3PlanePelPtr& operator++();
const cc3PlanePelPtr operator++(int);
```

- ```
  cc3PlanePelPtr& operator++();
  ```

  Implements the prefix **++** operator for this class.

- ```
  const cc3PlanePelPtr operator++(int);
  ```

  Implements the postfix **++** operator for this class.

operator--

```
cc3PlanePelPtr& operator--();
const cc3PlanePelPtr operator--(int);
```

- ```
  cc3PlanePelPtr& operator--();
  ```

  Implements the prefix **– –** operator for this class.

- ```
  const cc3PlanePelPtr operator--(int);
  ```

  Implements the postfix **– –** operator for this class.

operator[]

```
cc3PlanePelRef operator[] (int d) const;
```

Implements the subscript operator for this class.

cc3PlanePelPtr

operator* `cc3PlanePelRef operator*() const;`

Implements the pointer dereferencing operator for this class.

operator-> `cc3PlanePelRef *operator->() const;`

Implements the member dereferencing operator for this class.

cc3PlanePelPtr_const

```
#include <ch_cvl/colorpel.h>

class cc3PlanePelPtr_const : public cc_3Ptrs;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **cc3PlanePelPtr_const** class provides **const** pointer semantics for the three-plane pixel class **cc3PlanePel** (see page 145). It is used by the **ccPelTraits** class (see page 2403).

Constructors/Destructors

cc3PlanePelPtr_const

```
cc3PlanePelPtr_const();

cc3PlanePelPtr_const(const c_UInt8 *ch1,
  const c_UInt8 *ch2, const c_UInt8 *ch3);

explicit cc3PlanePelPtr_const(const cc3PlanePel *pelPtr);
```

- `cc3PlanePelPtr_const();`

  Creates a three-plane pixel **const** pointer that does not point to any pixel.

- ```
  cc3PlanePelPtr_const(const c_UInt8 *ch1,
    const c_UInt8 *ch2, const c_UInt8 *ch3);
  ```

  Creates a three-plane pixel **const** pointer that points to the three values supplied.

  Parameters

ch1	A pointer to the plane 0 value.
ch2	A pointer to the plane 1 value
ch3	A pointer to the plane 2 value.
  ```

- `explicit cc3PlanePelPtr_const(const cc3PlanePel *pelPtr);`

  Creates a three-plane pixel **const** pointer that points to the same three-plane pixel pointed to by *pelPtr*.

  Parameters

pelPtr	A pointer to a three-plane pixel object.

Public Member Functions

pPlane0 `const c_UInt8* pPlane0() const;`

Returns a read-only pointer to the plane 0 element of the pixel.

pPlane1 `const c_UInt8* pPlane1();`

Returns a read-only pointer to the plane 1 element of the pixel.

pPlane2 `const c_UInt8* pPlane2() const;`

Returns a read-only pointer to the plane 2 element of the pixel.

Operators

These operators allow pointer arithmetic on pointers to three-plane pixels. As for pointers to scalars, the pixels are expected to be contiguous.

operator+ `cc3PlanePelPtr_const operator+(int d) const;`

Returns a read-only pointer to the *d*th pixel beyond this one.

Parameters

d	The distance from this pixel.

operator+= `cc3PlanePelPtr_const& operator+=(int d);`

Returns a reference the *d*th pixel beyond this one.

Parameters

d	The distance from this pixel.

operator- `cc3PlanePelPtr_const operator-;`

Returns a read-only pointer to the *d*th pixel before this one.

Parameters

d The distance from this pixel.

operator-=

```
cc3PlanePelPtr_const& operator-=(int d)
```

Returns a reference the *d*th pixel before this one.

Parameters

d The distance from this pixel.

operator-

```
int operator-(const cc3PlanePelPtr_const &r);
```

Returns the number of pixels between this pixel and another pixel.

Parameters

r A reference to the other pixel.

operator++

```
cc3PlanePelPtr_const& operator++();
const cc3PlanePelPtr_const operator++(int);
```

- ```
  cc3PlanePelPtr_const& operator++();
  ```

  Implements the prefix **++** operator for this class.

- ```
  const cc3PlanePelPtr_const operator++(int);
  ```

  Implements the postfix **++** operator for this class.

operator--

```
cc3PlanePelPtr_const& operator--();
const cc3PlanePelPtr_const operator--(int);
```

- ```
  cc3PlanePelPtr_const& operator--();
  ```

  Implements the prefix **– –** operator for this class.

- ```
  const cc3PlanePelPtr_const operator--(int);
  ```

  Implements the postfix **– –** operator for this class.

cc3PlanePelPtr_const

operator[]

```
cc3PlanePelRef_const operator[] (int d) const;
```

Implements the subscript operator for this class.

operator*

```
cc3PlanePelRef_const operator*() const;
```

Implements the pointer dereferencing operator for this class.

operator->

```
cc3PlanePelRef_const *operator->() const;
```

Implements the member dereferencing operator for this class.

operator==

```
bool operator==(const cc3PlanePelPtr_const &other) const;
```

Returns true if the three-plane pixel that this object points to has the same values as the three-plane pixel that *other* points to.

Parameters

other The pointer to the other three-plane pixel.

operator!=

```
bool operator!=(const cc3PlanePelPtr_const &other) const;
```

Returns true if the three-plane pixel that this object points to does not have the same values as the three-plane pixel that *other* points to.

Parameters

other The pointer to the other three-plane pixel.

operator const void*

```
operator const void*() const;
```

Returns this pointer only if none of the planes is null.

operator!

```
bool operator!() const;
```

Returns true if one of the planes is null.

cc3PlanePelRef

```
#include <ch_cvl/colorpel.h>

class cc3PlanePelRef : public cc3PlanePelRef_const;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **cc3PlanePelRef** class provides reference semantics for the three-plane pixel class **cc3PlanePel** (see page 145). It is used by the **ccPelTraits** class (see page 2403).

Constructors/Destructors

cc3PlanePelRef `cc3PlanePelRef(const cc3PlanePel& pel);`

Creates a reference to a three-plane pixel.

Parameters

pel The pixel.

Public Member Functions

plane0 `c_UInt8& plane0() const;`

Returns a read-write reference to the plane 0 element of the pixel.

plane1 `c_UInt8& plane1() const;`

Returns a read-write reference to the plane 1 element of the pixel.

plane2 `c_UInt8& plane2() const;`

Returns a read-write reference to the plane 1 element of the pixel.

Operators

operator&

```
cc3PlanePelPtr operator&() const;
```

Returns a pointer class object referring to the same byte locations as this object.

operator=

```
cc3PlanePelRef operator=
   (const cc3PlanePelRef_const& cRef) const;

cc3PlanePelRef operator= (const cc3PlanePelRef& ref) const;

cc3PlanePelRef operator= (const cc3PlanePel& p) const;
```

Assignment operator.

The semantics are different from compiler-generated assignment operator. Rather than copying member objects, the byte locations pointed to by the member objects (which are of type **c_UInt8 ***) are copied. The semantics are consistent with the usage of this class as a proxy object (or reference class) to the **cc3PlanePel** class.

- ```
  cc3PlanePelRef operator=
     (const cc3PlanePelRef_const& cRef) const;
  ```

  Parameters

  cRef The const object to assign.

- ```
  cc3PlanePelRef operator= (const cc3PlanePelRef& ref) const;
  ```

  Parameters

  ref The object to assign.

- ```
  cc3PlanePelRef operator= (const cc3PlanePel& p) const;
  ```

  Parameters

  p The object to assign.
  ```

cc3PlanePelRef_const

```
#include <ch_cvl/colorpel.h>

class cc3PlanePelRef_const : public cc_3Ptrs;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **cc3PlanePelRef_const** class provides **const** reference semantics for the three-plane pixel class **cc3PlanePel** (see page 145). It is used by the **ccPelTraits** class (see page 2403).

Constructors/Destructors

cc3PlanePelRef_const

```
cc3PlanePelRef_const(const cc3PlanePel& pel);
```

Creates a **const** reference to a three-plane pixel.

Parameters

pel The pixel.

Public Member Functions

plane0

```
const c_UInt8& plane0();
```

Returns a read-only reference to the plane 0 element of the pixel.

plane1

```
const c_UInt8& plane1() const;
```

Returns a read-only reference to the plane 1 element of the pixel.

plane2

```
const c_UInt8& plane2() const;
```

Returns a read-only reference to the plane 2 element of the pixel.

Operators

operator==

```
bool operator==(const cc3PlanePel& p) const;
bool operator==(const cc3PlanePelRef_const& p) const;
```

- `bool operator==(const cc3PlanePel& p) const;`

  Returns true if the pixel referred to by this object has the same values as the pixel *p*.

  Parameters

  p The three-plane pel to compare to.

- `bool operator==(const cc3PlanePelRef_const& p) const;`

  Returns true if the pixel referred to by this object has the same values as the pixel *p*.

  Parameters

  p The three-plane pel to compare to.

operator!=

```
bool operator!=(const cc3PlanePel& p) const;
bool operator!=(const cc3PlanePelRef_const& p) const;
```

- `bool operator!=(const cc3PlanePel& p) const;`

  Returns true if the pixel referred to by this object does not have the same values as the pixel *p*.

  Parameters

  p The three-plane pel to compare to.

- `bool operator!=(const cc3PlanePelRef_const& p) const;`

  Returns true if the pixel referred to by this object does not have the same values as the pixel *p*.

  Parameters

  p The three-plane pel to compare to.

operator&

```
cc3PlanePelPtr_const operator&() const;
```

Returns a pointer class object referring to the same byte locations as this object.

cc3Vect

```
#include <ch_cvl/vector.h>

class cc3Vect;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a three-dimensional vector, a quantity that is determined by its length (magnitude) and direction. Throughout the CVL, **cc3Vector** is used to describe points in three-dimensional space, so you can also think of a vector as an (x,y,z) coordinate.

Notes

cc3Vect is the name of the class as it is used throughout the Cognex Vision Library. This name is actually a **typedef** for **ccVector<3>**. Both forms are valid and entirely equivalent.

Do not confuse the CVL vectors classes **cc2Vect** and **cc3Vect** with the C++ Standard Template Library's **vector** template class. Though both have the same mathematical underpinnings, the CVL vector classes are generally used to describe the location of points while the STL **vector** class is used to implement flexible arrays.

Constructors/Destructors

cc3Vect

```
cc3Vect();

cc3Vect(double x, double y, double z);
```

- `cc3Vect();`

  The default constructor creates a vector in which all three components are set to zero.

- `cc3Vect(double x, double y, double z);`

  Creates a vector with the specified components.

Parameters

x	The x-component of the vector.
y	The y-component of the vector.
z	The z-component of the vector.

Operators

operator[]

```
double& operator[](int d);
double operator[](int d) const;
```

- `double& operator[](int d);`

Provides array-like access to the vector. Typically you use this form when your application works with both two- and three-dimensional vectors. This form allows you to set the value of a component.

Parameters

d	The index of the element to return.
	0: The x-component.
	1: The y-component.
	2: The z-component.

- `double operator[](int d) const;`

Provides array-like access to the vector. Typically you use this form when your application works with both two- and three-dimensional vectors.

Parameters

d	The index of the element to return.
	0: The x-component.
	1: The y-component.
	2: The z-component.

operator+

```
cc3Vect operator+(const cc3Vect& v) const;
```

Returns a vector that is the result of adding this vector and another vector.

Parameters

v The vector to add to this vector.

operator+=

```
cc3Vect& operator+=(const cc3Vect& v);
```

Adds this vector to another vector and returns the result.

Parameters

v The vector to add to this vector.

operator-

```
cc3Vect operator-() const;
cc3Vect operator-(const cc3Vect& v) const;
```

- ```
  cc3Vect operator-() const;
  ```

  The unary minus operator.

- ```
  cc3Vect operator-(const cc3Vect& v) const;
  ```

  Returns a vector that is the result of subtracting the vector v from this vector.

  Parameters

  v The vector to subtract from this vector.

operator-=

```
cc3Vect& operator-=(const cc3Vect& v);
```

Subtracts the vector v from this vector and returns the result.

Parameters

v The vector to subtract from this vector.

operator*

```
cc3Vect operator*(double d) const;
friend cc3Vect operator*(double d, const cc3Vect& v);
double operator*(const cc3Vect& v) const;
```

- ```
  cc3Vect operator*(double d) const;
  ```

  Returns a vector that is the result of multiplying each element in the vector by *d*.

  Parameters

  d The value by which to multiply each element of the vector.
  ```

- `friend cc3Vect operator*(double d, const cc3Vect& v);`

  Returns a vector that is the result of multiplying each element in the vector by *d*.

  Parameters

d	The value by which to multiply each element of the vector.
v	The vector.

- `double operator*(const cc3Vect& v) const;`

  Returns the dot product of this vector and the vector *v*.

  Parameters

v	The other vector.

operator*= `cc3Vect& operator*=(double d);`

Multiplies each element in this vector by *d* and returns the result.

Parameters

d	The value by which to multiply each element of the vector.

operator/ `cc3Vect operator/(double d) const;`

Returns a vector that is the result of dividing each element in this vector by *d*.

Parameters

d	The value by which to divide each element of the vector.

operator/= `cc3Vect& operator/=(double d) const;`

Divides each element in this vector by *d* and returns the result.

Parameters

d	The value by which to divide each element of the vector.

operator== `bool operator==(const cc3Vect& v) const;`

Returns true if this vector is equal to another vector.

Parameters

v	The other vector.

operator!=

```
bool operator!=(const cc3Vect& v) const;
```

Returns true if this vector is not equal to another vector.

Parameters

v	The other vector.

Public Member Functions

x

```
double x() const;
void x(double newX);
```

- ```
  double x() const;
  ```

  Returns the vector's x-component.

- ```
  void x(double newX);
  ```

  Sets the vector's x-component.

  Parameters

newX	The new x-component.

y

```
double y() const;
void y(double newY);
```

- ```
  double y() const;
  ```

  Returns the vector's y-component.

- ```
  void y(double newY);
  ```

  Sets the vector's y-component.

  Parameters

newY	The new y-component.

z

```
double z() const;
void z(double newZ);
```

- `double z() const;`

  Returns the vector's z-component.

- `void z(double newZ);`

  Sets the vector's z-component.

  Parameters

  newZ The new z-component.

len

```
double len() const;
```

Returns the length (or magnitude) of the vector.

isNull

```
bool isNull() const;
```

Returns true if this is a null vector (all components are zero).

unit

```
cc3Vect unit() const;
```

Returns a unit vector parallel to this vector. A unit vector is a vector whose length is 1.

Throws

ccMathError::NullVector
The x-component, the y-component, and the z-component are zero.

project

```
cc3Vect project(const cc3Vect&) const;
```

Returns a vector that is the result of projecting the vector *v* onto this vector.

Parameters

v The vector to project onto this vector.

Throws

ccMathError::NullVector
The x-component, the y-component, and the z-component are zero.

An error is not thrown if *v* is null or the two vectors are perpendicular to each other.

distance

```
double distance(const cc3Vect& v) const;
```

Returns the distance to the other vector. This is the length of difference between this vector and vector *v*.

Parameters

v	The other vector.

dot

```
double dot(const cc3Vect& v) const;
```

Returns the dot product of this vector and the vector *v*.

Parameters

v	The other vector.

cross

```
cc3Vect cross(const cc3Vect& v) const;
```

Returns the cross product of this vector and *v*.

Parameters

v	The other vector.

cc3Vect

cc8500l

```
#include <ch_cvl/vp8500l.h>

class cc8500l : public ccUnknownFG,
                public ccUnknownPIO;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes the Cognex MVS-8500L frame grabber. There is one instance of this class for each MVS-8500L board in a system. When you write your application you create the frame grabber object you will use, with code similar to the following:

```
cc8500l& fg = cc8500l::get(i);
```

In this example, *i* is the MVS-8500L board number in your system, starting with 0. If you have only one MVS-8500L in your system *i* will always be 0. If you have more than one MVS-8500L board in your system you will need to know the board number you wish to use so you can code the above line properly.

Constructors/Destructors

A single instance of this class is created automatically for each MVS-8500L installed in your system.

Public Member Functions

inputLine

```
virtual ccInputLine inputLine(c_Int32 line);
```

Returns a **ccInputLine** object that represents the specified logical input line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line number. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

The MVS-8500L provides eight bidirectional TTL lines. Each line can be independently enabled as an input or output line using **ccOutputLine::enable()** or **ccInputLine::enable()**.

The eight lines are numbered 8 through 15.

At power-on of the MVS-8500L, all lines are enabled as input lines; they must be explicitly enabled for use as output lines. At power-on, all lines are pulled high by an internal pull-up resistor.

The MVS-8500L's parallel I/O lines have three connection options. They can be used as TTL lines, can be converted to opto-isolated line pairs, or a combination of TTL and opto-isolated lines. These connection options are described in the *MVS-8500 Hardware Manual.* Consult that manual for the correspondence between the signal names used in the table below and the pin numbers to which you connect device wiring.

Line 8 has a dual purpose as either programmable TTL line or as a trigger line for the currently active camera. If the acquisition FIFO's **triggerEnable()** function is invoked for line 8, then line 8 is not available for use as an input line with this function.

Similarly, line 9 has a dual purpose as either programmable TTL line or strobe control line for the currently active camera. If the acquisition FIFO's **strobeEnable()** function is invoked for line 9, then line 9 is not available for use as an input line with this function.

The following table shows the logical *line* numbers for the eight parallel I/O lines on the MVS-8500L, when each bidirectional line is configured as an input line.

inputLine *line* parameter	**Signal name on MVS-8500L**	**Device Connection**
8	TTL_BI_8	Trigger for the currently active camera, if enabled. Otherwise a bidirectional line, controls any TTL input device when this line is enabled for input.
9	TTL_BI_9	Strobe for the currently active camera, if enabled. Otherwise a bidirectional line, controls any TTL input device when this line is enabled for input.
10	TTL_BI_10	Bidirectional line, controls any TTL input device when this line is enabled for input
11	TTL_BI_11	Bidirectional, as above

inputLine *line* parameter	Signal name on MVS-8500L	Device Connection
12	TTL_BI_12	Bidirectional, as above
13	TTL_BI_13	Bidirectional, as above
14	TTL_BI_14	Bidirectional, as above
15	TTL_BI_15	Bidirectional, as above

The *line* numbers in the table above are the same whether devices are wired with the TTL, opto-isolated, or combination connection options. Consult the *MVS-8500 Hardware Manual* for the pin numbers for each connection option that correspond to the signal names in the table above.

outputLine

```
virtual ccOutputLine outputLine(c_Int32 line);
```

Returns a **ccOutputLine** object that represents the specified logical output line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

See the *Notes* above for **inputLine()**. Remember that at power-on of the MVS-8500L, all lines are enabled as input lines; they must be explicitly enabled for use as output lines, using **ccOutputLine::enable()**.

The eight lines are numbered 8 through 15.

Consult the *MVS-8500 Hardware Manual* for the correspondence between the signal names used in the table below and the pin numbers to which you connect device wiring.

The following table shows the logical *line* numbers for the eight parallel I/O lines on the MVS-8500L, when each bidirectional line is configured as an output line.

outputLine *line* parameter	**Signal name on MVS-8500L**	**Device Connection**
8	TTL_BI_8	Trigger for the currently active camera, if enabled. Otherwise a bidirectional line, controls any TTL input device when this line is enabled for input.
9	TTL_BI_9	Strobe for the currently active camera, if enabled. Otherwise a bidirectional line, controls any TTL input device when this line is enabled for input.
10	TTL_BI_10	Bidirectional line, controls any TTL input device when this line is enabled for input
11	TTL_BI_11	Bidirectional, as above
12	TTL_BI_12	Bidirectional, as above
13	TTL_BI_13	Bidirectional, as above
14	TTL_BI_14	Bidirectional, as above
15	TTL_BI_15	Bidirectional, as above

The *line* numbers in the table above are the same whether devices are wired with the TTL, opto-isolated, or combination connection options. Consult the *MVS-8500 Hardware Manual* for the pin numbers for each connection option that correspond to the signal names in the table above.

numInputLines `virtual c_Int32 numInputLines() const;`

Returns the number of total input lines for this hardware. The total depends upon the configuration set by **setIOConfig()**.

This is an override from class **ccParallelIO**.

numOutputLines `virtual c_Int32 numOutputLines() const;`

Returns the number of total output lines for this hardware.

This is an override from class **ccParallelIO**.

getIOConfig `virtual ccIOConfig& getIOConfig() const;`

Returns the parallel I/O board configuration.

This is an override from class **ccParallelIO**.

setIOConfig `virtual void setIOConfig(ccIOConfig& config);`

Sets the parallel I/O board configuration. See **ccIOConfig** and its derived classes, as discussed in **ccIOConfig** on page 1799 and in *ch_cvl/pioconfg.h*.

This is an override from class **ccParallelIO**.

Parameters

config The I/O board configuration.

Notes

The *config* parameter is one of the following classes derived from **ccIOConfig**:

ccIO8500l (for use with cable 300-0390)
ccIOExternal8500l (for use with cable 300-0389)
ccIOSplit8500l (for use with cable 300-0399)

Example

This example sets the I/O configuration for an MVS-8500L frame grabber using cable 300-0390 and the pass-through TTL connection module.

```
cc8500l& fg = cc8500l::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIO8500l());
```

Static Functions

count

```
static c_Int32 count();
```

Returns the number of Cognex MVS-8500L frame grabbers installed in the system.

get

```
static cc8500l& get(c_Int32 i = 0);
```

Returns the **cc8500l** object associated with the Cognex MVS-8500L frame grabber whose index is *i*. Index numbers begin with 0. See the introduction to this reference page for additional information.

Parameters

i The Cognex MVS-8500L board index.

Throws

ccBoard::BadParams
If *i* is not a valid index.

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding
The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse
The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error,

a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8100m**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occured while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

cc8500l

cc8501

```
#include <ch_cvl/vp8501.h>

class cc8501 : public ccBoard,
               public ccFrameGrabber,
               public ccParallelIO;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes the Cognex MVS-8501 frame grabber and the Cognex MVS-8511 (and MVS-8511e) frame grabber.

Note The information contained in this section for Cognex MVS-8511 frame grabbers is valid for Cognex MVS-8511e frame grabbers.

There is one instance of this class for each MVS-8501 or MVS-8511 board in a system. When you write your application you create the frame grabber object you will use, with code similar to the following:

```
cc8501& fg = cc8501::get(i);
```

In this example, *i* is the MVS-8501 board number in your system, starting with 0. If you have only one MVS-8501 in your system *i* will always be 0. If you have more than one MVS-8501 board in your system you will need to know the board number you wish to use so you can code the above line properly.

Generally, MVS-8501 board 0 is the MVS-8501 board in the lower numbered PCI slot. However, this can vary with different motherboard manufacturers, and can be BIOS dependent. Consult your system hardware documentation to verify the system's PCI slot numbering system when you use more than one MVS-8501 or MVS-8511 in your system.

Constructors/Destructors

A single instance of this class is created automatically for each MVS-8501 and MVS-8511 installed in your system.

Public Member Functions

inputLine

```
virtual ccInputLine inputLine(c_Int32 line);
```

Returns a **ccInputLine** object that represents the specified logical input line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line number. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

The MVS-8501 or MVS-8511 provide sixteen bidirectional TTL lines. On some Cognex frame grabber families, the input or output direction of all bidirectional lines is set as a group. By contrast, on the MVS-8500 series, each line can be independently enabled as an input or output line using **ccOutputLine::enable()** or **ccInputLine::enable()**.

At power-on of the MVS-8501 or MVS-8511, all sixteen lines are enabled as input lines; they must be explicitly enabled for use as output lines. At power-on, all lines are pulled high by an internal pull-up resistor. By contrast, on other Cognex frame grabbers, parallel I/O lines are pulled low or allowed to float.

The MVS-8501's or MVS-8511's parallel I/O lines have three connection options. They can be used as TTL lines, can be converted to opto-isolated line pairs, or a combination of TTL and opto-isolated lines. These connection options are

described in the *MVS-8500 Hardware Manual*. Consult that manual for the correspondence between the signal names used in the table below and the pin numbers to which you connect device wiring.

Line 8 has a dual purpose as either programmable TTL line or as a trigger line for the currently active camera. If the acquisition FIFO's **triggerEnable()** function is invoked for line 8, then line 8 is not available for use as an input line with this function.

Similarly, line 9 has a dual purpose as either programmable TTL line or strobe control line for the currently active camera. If the acquisition FIFO's **strobeEnable()** function is invoked for line 9, then line 9 is not available for use as an input line with this function.

The following table shows the logical *line* numbers for the sixteen parallel I/O lines on the MVS-8501 or MVS-8511, when each bidirectional line is configured as an input line.

inputLine *line* parameter	**Signal name on MVS-8501**	**Device Connection**
0	TTL_BI_0	Bidirectional line, controls any TTL input device when this line is enabled for input
1	TTL_BI_1	Bidirectional, as above
2	TTL_BI_2	Bidirectional, as above
3	TTL_BI_3	Bidirectional, as above
4	TTL_BI_4	Bidirectional, as above
5	TTL_BI_5	Bidirectional, as above
6	TTL_BI_6	Bidirectional, as above
7	TTL_BI_7	Bidirectional, as above
8	TTL_BI_8	Trigger for the currently active camera, if enabled, or bidirectional, as above
9	TTL_BI_9	Strobe for the currently active camera, if enabled, or bidirectional, as above
10	TTL_BI_10	Bidirectional line, controls any TTL input device when this line is enabled for input
11	TTL_BI_11	Bidirectional, as above

inputLine *line* parameter	Signal name on MVS-8501	Device Connection
12	TTL_BI_12	Bidirectional, as above
13	TTL_BI_13	Bidirectional, as above
14	TTL_BI_14	Bidirectional, as above
15	TTL_BI_15	Bidirectional, as above

The *line* numbers in the table above are the same whether devices are wired with the TTL, opto-isolated, or combination connection options. Consult the *MVS-8500 Hardware Manual* for the pin numbers for each connection option that correspond to the signal names in the table above.

outputLine

```
virtual ccOutputLine outputLine(c_Int32 line);
```

Returns a **ccOutputLine** object that represents the specified logical output line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

See the *Notes* above for **inputLine()**. Remember that at power-on of the MVS-8501 or MVS-8511, all sixteen lines are enabled as input lines; they must be explicitly enabled for use as output lines, using **ccOutputLine::enable()**.

Consult the *MVS-8500 Hardware Manual* for the correspondence between the signal names used in the table below and the pin numbers to which you connect device wiring.

The following table shows the logical *line* numbers for the sixteen parallel I/O lines on the MVS-8501 or MVS-8511, when each bidirectional line is configured as an output line.

outputLine *line* parameter	**Signal name on MVS-8501**	**Device Connection**
0	TTL_BI_0	Bidirectional line, controls any TTL output device when this line is enabled for output
1	TTL_BI_1	Bidirectional, as above
2	TTL_BI_2	Bidirectional, as above
3	TTL_BI_3	Bidirectional, as above
4	TTL_BI_4	Bidirectional, as above
5	TTL_BI_5	Bidirectional, as above
6	TTL_BI_6	Bidirectional, as above
7	TTL_BI_7	Bidirectional, as above
8	TTL_BI_8	Trigger for the currently active camera if enabled, or bidirectional, as above
9	TTL_BI_9	Strobe for the currently active camera if enabled, or bidirectional, as above
10	TTL_BI_10	Bidirectional line, controls any TTL output device when this line is enabled for output
11	TTL_BI_11	Bidirectional, as above
12	TTL_BI_12	Bidirectional, as above
13	TTL_BI_13	Bidirectional, as above

outputLine *line* parameter	Signal name on MVS-8501	Device Connection
14	TTL_BI_14	Bidirectional, as above
15	TTL_BI_15	Bidirectional, as above

The *line* numbers in the table above are the same whether devices are wired with the TTL, opto-isolated, or combination connection options. Consult the *MVS-8500 Hardware Manual* for the pin numbers for each connection option that correspond to the signal names in the table above.

numInputLines

```
virtual c_Int32 numInputLines() const;
```

Returns the number of total input lines for this hardware. The total depends upon the configuration set by **setIOConfig()**.

This is an override from class **ccParallelIO**.

numOutputLines

```
virtual c_Int32 numOutputLines() const;
```

Returns the number of total output lines for this hardware.

This is an override from class **ccParallelIO**.

getIOConfig

```
virtual ccIOConfig& getIOConfig() const;
```

Returns the parallel I/O board configuration.

This is an override from class **ccParallelIO**.

setIOConfig

```
virtual void setIOConfig(ccIOConfig& config);
```

Sets the parallel I/O board configuration. See **ccIOConfig** and its derived classes, as discussed in **ccIOConfig** on page 1799 and in *ch_cvl/pioconfg.h*.

This is an override from class **ccParallelIO**.

Parameters

config The I/O board configuration.

Notes

The *config* parameter is one of the following classes derived from **ccIOConfig**:

ccIO8501 (for use with cable 300-0390)
ccIOExternal8501 (for use with cable 300-0389)
ccIOSplit8501 (for use with cable 300-0399)

Example

This example sets the I/O configuration for an MVS-8501 frame grabber using cable 300-0390 and the pass-through TTL connection module.

```
cc8501& fg = cc8501::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIO8501());
```

is8510

```
bool is8510() const;
```

Returns whether the 'gotten' object returned by **cc8501::get(*i*)** is an 8501 or 8511 frame grabber.

Notes

This function offers an alternate approach to differentiating by means of the grabber's 'pretty' name, for example, “8501” versus “8511”.

Static Functions

count

```
static c_Int32 count();
```

Returns the number of Cognex MVS-8501 and MVS-8511 frame grabbers installed in the system.

get

```
static cc8501& get(c_Int32 i = 0);
```

Returns the **cc8501** object associated with the Cognex MVS-8501 or MVS-8511 frame grabber whose index is *i*. Index numbers begin with 0. See the introduction to this reference page for additional information.

Parameters

i The Cognex MVS-8501 or MVS-8511 board index.

Throws

ccBoard::BadParams
If *i* is not a valid index.

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding

The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse

The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error, a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8100m**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occured while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

cc8504

```
#include <ch_cvl/vp8504.h>

class cc8504 : public ccBoard,
               public ccFrameGrabber,
               public ccParallelIO;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes the Cognex MVS-8504 (and MVS-8504e) frame grabber and the Cognex MVS-8514 (and MVS-8514e) frame grabber.

Note The information contained in this section for Cognex MVS-8504 and MVS-8514 frame grabbers is valid for Cognex MVS-8504e and MVS-8514e frame grabbers respectively.

There is one instance of this class for each MVS-8504 and MVS-8514 board in a system. When you write your application you create the frame grabber object you will use, with code similar to the following:

```
cc8504& fg = cc8504::get(i);
```

In this example, *i* is the MVS-8504 board number in your system, starting with 0. If you have only one MVS-8504 in your system *i* will always be 0. If you have more than one MVS-8504 board in your system you will need to know the board number you wish to use so you can code the above line properly.

Generally, MVS-8504 (or MVS-8514) board 0 is the MVS-8504 (or MVS-8514) board in the lower numbered PCI slot. However, this can vary with different motherboard manufacturers, and can be BIOS dependent. Consult your system hardware documentation to verify the system's PCI slot numbering system when you use more than one MVS-8504 or MVS-8514 in your system.

Constructors/Destructors

A single instance of this class is created automatically for each MVS-8504 and MVS-8514 installed in your system.

Public Member Functions

inputLine

```
virtual ccInputLine inputLine(c_Int32 line);
```

Returns a **ccInputLine** object that represents the specified logical input line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line number. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

The MVS-8504 or MVS-8514 provide sixteen bidirectional TTL lines. On some Cognex frame grabber families, the input or output direction of all bidirectional lines is set as a group. By contrast, on the MVS-8500 series, each line can be independently enabled as an input or output line using **ccOutputLine::enable()** or **ccInputLine::enable()**.

At power-on of the MVS-8504 or MVS-8514, all sixteen lines are enabled as input lines; they must be explicitly enabled for use as output lines. At power-on, all lines are pulled high by an internal pull-up resistor. By contrast, on other Cognex frame grabbers, parallel I/O lines are pulled low or allowed to float.

The MVS-8504's or MVS-8514's parallel I/O lines have three connection options. They can be used as TTL lines, can be converted to opto-isolated line pairs, or a combination of TTL and opto-isolated lines. These connection options are

described in the *MVS-8500 Hardware Manual*. Consult that manual for the correspondence between the signal names used in the table below and the pin numbers to which you connect device wiring.

Lines 8, 10, 12, and 14 have a dual purpose as either programmable TTL lines or dedicated trigger lines. If the acquisition FIFO's **triggerEnable()** function is invoked for a given camera channel, the dedicated trigger line for that channel is not available for use as an input line with this function.

Similarly, lines 9, 11, 13, and 15 have a dual purpose as either programmable TTL lines or dedicated strobe control lines. If the acquisition FIFO's **strobeEnable()** function is invoked for a given camera channel, the dedicated strobe line for that channel is not available for use as an input line with this function.

The following table shows the logical *line* numbers for the sixteen parallel I/O lines on the MVS-8504 or MVS-8514, when each bidirectional line is configured as an input line.

inputLine *line* parameter	**Signal name on MVS-8504**	**Device Connection**
0	TTL_BI_0	Bidirectional line, controls any TTL input device when this line is enabled for input
1	TTL_BI_1	Bidirectional, as above
2	TTL_BI_2	Bidirectional, as above
3	TTL_BI_3	Bidirectional, as above
4	TTL_BI_4	Bidirectional, as above
5	TTL_BI_5	Bidirectional, as above
6	TTL_BI_6	Bidirectional, as above
7	TTL_BI_7	Bidirectional, as above
8	TTL_BI_8	Trigger for camera channel 1 if enabled, or bidirectional, as above
9	TTL_BI_9	Strobe for camera channel 1 if enabled, or bidirectional, as above
10	TTL_BI_10	Trigger for camera channel 2 if enabled, or bidirectional, as above
11	TTL_BI_11	Strobe for camera channel 2 if enabled, or bidirectional, as above

inputLine *line* parameter	Signal name on MVS-8504	Device Connection
12	TTL_BI_12	Trigger for camera channel 3 if enabled, or bidirectional, as above
13	TTL_BI_13	Strobe for camera channel 3 if enabled, or bidirectional, as above
14	TTL_BI_14	Trigger for camera channel 4 if enabled, or bidirectional, as above
15	TTL_BI_15	Strobe for camera channel 4 if enabled, or bidirectional, as above

The *line* numbers in the table above are the same whether devices are wired with the TTL, opto-isolated, or combination connection options. Consult the *MVS-8500 Hardware Manual* for the pin numbers for each connection option that correspond to the signal names in the table above.

outputLine

```
virtual ccOutputLine outputLine(c_Int32 line);
```

Returns a **ccOutputLine** object that represents the specified logical output line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

See the *Notes* above for **inputLine()**. Remember that at power-on of the MVS-8504, all sixteen lines are enabled as input lines; they must be explicitly enabled for use as output lines, using **ccOutputLine::enable()**.

Consult the *MVS-8500 Hardware Manual* for the correspondence between the signal names used in the table below and the pin numbers to which you connect device wiring.

The following table shows the logical *line* numbers for the sixteen parallel I/O lines on the MVS-8504 or MVS-8514, when each bidirectional line is configured as an output line.

outputLine *line* parameter	**Signal name on MVS-8504**	**Device Connection**
0	TTL_BI_0	Bidirectional line, controls any TTL output device when this line is enabled for output
1	TTL_BI_1	Bidirectional, as above
2	TTL_BI_2	Bidirectional, as above
3	TTL_BI_3	Bidirectional, as above
4	TTL_BI_4	Bidirectional, as above
5	TTL_BI_5	Bidirectional, as above
6	TTL_BI_6	Bidirectional, as above
7	TTL_BI_7	Bidirectional, as above
8	TTL_BI_8	Trigger for camera channel 1 if enabled, or bidirectional, as above
9	TTL_BI_9	Strobe for camera channel 1 if enabled, or bidirectional, as above
10	TTL_BI_10	Trigger for camera channel 2 if enabled, or bidirectional, as above
11	TTL_BI_11	Strobe for camera channel 2 if enabled, or bidirectional, as above
12	TTL_BI_12	Trigger for camera channel 3 if enabled, or bidirectional, as above

outputLine *line* parameter	Signal name on MVS-8504	Device Connection
13	TTL_BI_13	Strobe for camera channel 3 if enabled, or bidirectional, as above
14	TTL_BI_14	Trigger for camera channel 4 if enabled, or bidirectional, as above
15	TTL_BI_15	Strobe for camera channel 4 if enabled, or bidirectional, as above

The *line* numbers in the table above are the same whether devices are wired with the TTL or opto-isolated connection options. Consult the *MVS-8500 Hardware Manual* for the pin numbers for each connection option that correspond to the signal names in the table above.

numInputLines `virtual c_Int32 numInputLines() const;`

Returns the number of total input lines for this hardware. The total depends upon the configuration set by **setIOConfig()**.

This is an override from class **ccParallelIO**.

numOutputLines `virtual c_Int32 numOutputLines() const;`

Returns the number of total output lines for this hardware.

This is an override from class **ccParallelIO**.

getIOConfig `virtual ccIOConfig& getIOConfig() const;`

Returns the parallel I/O board configuration.

This is an override from class **ccParallelIO**.

setIOConfig `virtual void setIOConfig(ccIOConfig& config);`

Sets the parallel I/O board configuration. See **ccIOConfig** and its derived classes, as discussed in **ccIOConfig** on page 1799 and in *ch_cvl/pioconfg.h*.

This is an override from class **ccParallelIO**.

Parameters

config The I/O cable and connection box configuration, in the form of a class derived from **ccIOConfig**.

Notes

The *config* parameter is one of the following classes derived from **ccIOConfig**:

ccIO8504 (for use with cable 300-0390)
ccIOExternal8504 (for use with cable 300-0389)
ccIOSplit8504 (for use with cable 300-0399)

Example

This example sets the I/O configuration for an MVS-8504 frame grabber using cable 300-0390 and the pass-through TTL connection module.

```
cc8504& fg = cc8504::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIO8504());
```

is8510

```
bool is8510() const;
```

Returns whether the 'gotten' object returned by **cc8504::get(*i*)** is an 8504 or 8514 frame grabber.

Notes

This function offers an alternate approach to differentiating by means of the grabber's 'pretty' name, for example, “8504” versus “8514”.

Static Functions

count

```
static c_Int32 count();
```

Returns the number of Cognex MVS-8504 frame grabbers and Cognex MVS-8514 frame grabbers installed in the system.

get

```
static cc8504& get(c_Int32 i = 0);
```

Returns the **cc8504** object associated with the Cognex MVS-8504 frame grabber or the Cognex MVS-8514 frame grabber whose index is *i*. Index numbers begin with 0. See the introduction to this reference page for additional information.

Parameters

i The Cognex MVS-8504 or MVS-8514 board index.

Throws

ccBoard::BadParams
If *i* is not a valid index.

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding

The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse

The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error, a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8100m**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occured while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

cc8600

```
#include <ch_cvl/vp8600.h>

class cc8600 : public ccBoard,
               public ccFrameGrabber,
               public ccParallelIO;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes the Cognex MVS-8600 and MVS-8600e frame grabbers. There is one instance of this class for each frame grabber board in a system.

The MVS-8600 and MVS-8600e series frame grabbers consist of three frame grabber boards that plug into your PC. The MVS-8601 and MVS-8602 plug into the PCI bus and the MVS-8602e plugs into the PCI Express bus. The MVS-8601 has one camera port and can support one camera. The MVS-8602 and MVS-8602e have two camera ports and can support one or two cameras each. In this class reference we use the term MVS-8600, or just 8600, to refer to all three of the MVS-8600 and MVS-8600e frame grabber boards.

When you write your application, you create the frame grabber object you will use with code like this example:

```
cc8600& fg = cc8600::get(i);
```

In this example, *i* is the MVS-8600 or MVS-8600e board number in your system, starting with 0. If you have only one MVS-8600 or MVS-8600e in your system *i* will always be 0. If you have more than one, you will need to know the board number you wish to address so you can code the above line properly.

Generally, board 0 is the MVS-8600 or MVS-8600e board in the lower numbered PCI slot. However, this can vary with different motherboard manufacturers, and can be BIOS dependent. Consult your system hardware documentation to verify the system's PCI slot numbering system when you use more than one MVS-8600 or MVS-8600e in your system.

Constructors/Destructors

A single instance of this class is created automatically for each MVS-8600 installed in your system.

Public Member Functions

inputLine

```
virtual ccInputLine inputLine(c_Int32 line);
```

Returns a **ccInputLine** object that represents the specified logical input line. This is an override from class **ccParallelIO**.

Parameters

line — The logical line number. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

The MVS-8600 provides six or eight pairs of opto-isolated input lines, depending on the I/O configuration you specify with **setIOConfig()**.

I/O Configuration	Number of programmable input lines
ccIO8600LVDS	8
ccIO8600TTL	8
ccIO8600DualLVDS	6

The following table shows the logical *line* numbers for the eight opto-isolated input line pairs. Match the signal names to connection pin numbers by using the *MVS-8600 Hardware Manual*.

inputLine *line* parameter	Signal name on MVS-8600	Notes
0	OPTO_IN_0±	Not available when connecting two line scan cameras with two LVDS encoders (that is, when using **ccIO8600DualLVDS**).
1	OPTO_IN_1±	
2	OPTO_IN_2±	
3	OPTO_IN_3±	
4	OPTO_IN_4±	
5	OPTO_IN_5±	
6	OPTO_IN_6±	
7	OPTO_IN_7±	

outputLine

```
virtual ccOutputLine outputLine(c_Int32 line);
```

Returns a **ccOutputLine** object that represents the specified logical output line. This is an override from class **ccParallelIO**.

Parameters

line The logical line. *line* must be a valid value for the configuration set by **setIOConfig()**.

Notes

The MVS-8600 provides six or eight pairs of opto-isolated output lines, depending on the I/O configuration you specify with **setIOConfig()**.

I/O Configuration	Number of programmable output lines
ccIO8600LVDS	8
ccIO8600TTL	8
ccIO8600DualLVDS	6

The following table shows the logical *line* numbers for the eight opto-isolated output line pairs. Match the signal names to connection pin numbers by using the *MVS-8600 Hardware Manual*.

outputLine *line* parameter	Signal name on MVS-8600	Notes
0	OPTO_OUT_0±	Not available when connecting two line scan cameras with two LVDS encoders (that is, when using **ccIO8600DualLVDS**).
1	OPTO_OUT_1±	
2	OPTO_OUT_2±	
3	OPTO_OUT_3±	
4	OPTO_OUT_4±	
5	OPTO_OUT_5±	
6	OPTO_OUT_6±	
7	OPTO_OUT_7±	

numInputLines `virtual c_Int32 numInputLines() const;`

Returns the number of total input lines for this hardware. The total depends upon the configuration set by **setIOConfig()**.

This is an override from class **ccParallelIO**.

numOutputLines `virtual c_Int32 numOutputLines() const;`

Returns the number of total output lines for this hardware.

This is an override from class **ccParallelIO**.

getIOConfig `virtual ccIOConfig& getIOConfig() const;`

Returns the parallel I/O board configuration.

This is an override from class **ccParallelIO**.

setIOConfig `virtual void setIOConfig(ccIOConfig& config);`

Sets the parallel I/O board configuration. See **ccIOConfig** and its derived classes, as discussed in **ccIOConfig** on page 1799 and in *ch_cvl/pioconfg.h*.

This is an override from class **ccParallelIO**.

Parameters

config	The I/O cable and connection box configuration, in the form of a class derived from **ccIOConfig**.

Notes

The *config* parameter is one of the following classes derived from **ccIOConfig**:

ccIO8600LVDS (for use with cable 300-0539)
ccIO8600TTL (for use with cable 300-0540)
ccIO8600DualLVDS (for use with cable 300-0538)

In all cases, the same I/O connection module is used, Cognex part number 800-5885-1.

Example

This example sets the I/O configuration for an MVS-8600 frame grabber using cable 300-0539 and the MVS-8600 I/O connection module.

```
cc8600& fg = cc8600::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio->setIOConfig(ccIO8600LVDS());
```

sizePelPool

```
c_Int32 sizePelPool();
```

```
void sizePelPool(c_Int32 desiredSize);
```

The pel pool is a special pool used to store images acquired using the MVS-8600 and MVS-8600e frame grabbers. Each frame grabber has its own pel pool. The default size is 32 MB each.

- `c_Int32 sizePelPool();`

  Returns the current pel pool size, in bytes.

- `void sizePelPool(c_Int32 desiredSize);`

  Sets a new pel pool size, in bytes.

  Parameters

  desiredSize The new pel pool size.

  Notes

  When you set a new pel pool size, you set the size for all MVS-8600 and MVS-8600e frame grabbers.

Static Functions

count

```
static c_Int32 count();
```

Returns the number of Cognex MVS-8600 frame grabbers installed in the system.

get

```
static cc8600& get(c_Int32 i = 0);
```

Returns the **cc8600** object associated with the Cognex MVS-8600 frame grabber whose index is *i*. Index numbers begin with 0. See the introduction to this reference page for additional information.

Parameters

i The Cognex MVS-8600 board index.

Throws

ccBoard::BadParams

If *i* is not a valid index.

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding

The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse

The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error, a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8600**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occurred while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from

an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

cc8BitInputLutProp

```
#include <ch_cvl/prop.h>

class cc8BitInputLutProp;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes the 8-bit lookup table, a 256-element array. The digitizer converts each pixel into an 8-bit value. This value is used as an index into the input lookup table. The resulting value is stored in the root image. For dual-tap cameras, which use one digitizer for the even field and another for the odd field, two lookup tables are used.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

cc8BitInputLutProp

```
cc8BitInputLutProp();

explicit cc8BitInputLutProp(const c_UInt8* ilut);

explicit cc8BitInputLutProp(const c_UInt8* ilut0,
  const c_UInt8* ilut1);
```

- `cc8BitInputLutProp();`

  Creates a new input lookup table property not associated with any FIFO. Uses *cc8BitInputLutProp::default8BitInputLut* as the lookup table.

- `explicit cc8BitInputLutProp(const c_UInt8* ilut);`

  Creates a new input lookup table property not associated with any FIFO. This function uses a copy of *ilut* as the lookup table.

Parameters

ilut — An array of 256 8-bit values (**c_UInt8**) to use as the lookup table.

- `explicit cc8BitInputLutProp(const c_UInt8* ilut0, const c_UInt8* ilut1);`

Creates a new input lookup table property not associated with any FIFO. This function uses a copy of *ilut0* as the lookup table for the fist (even) digitizer and *ilut1* as the lookup table for the second (odd) digitizer.

Parameters

ilut0 — An array of 256 8-bit values (**c_UInt8**) to use as the lookup table for the first (even) digitizer.

ilut1 — An array of 256 8-bit values (**c_UInt8**) to use as the lookup table for the second (odd) digitizer.

Enumerations

LutChannel

```
enum LutChannel;
```

This enumeration lets you specify the digitizer that an input lookup table refers to.

Value	Meaning
eOnly	The input lookup table is the only lookup table and applies to all digitizers.
eEven	The input lookup table applies to the digitizer for the even field. For single-tap cameras, *eEven* is the same as *eOnly*.
eOdd	The input lookup table applies to the digitizer for the odd field.

Public Member Functions

inputLut

```
void inputLut(const c_UInt8* ilut);
void inputLut(const c_UInt8* ilut, LutChannel lutCh);
const c_UInt8* inputLut(LutChannel lut=eOnly) const;
```

- `void inputLut(const c_UInt8* ilut);`

Sets the 8-bit input lookup table to a copy of *ilut*.

Parameters

ilut An array of 256 8-bit values (**c_UInt8**) to use as the lookup table.

- `void inputLut(const c_UInt8* ilut, LutChannel lutCh);`

  Sets the 8-bit lookup table for the specified digitizer to a copy of *ilut*.

  Parameters

  ilut An array of 256 8-bit values (**c_UInt8**) to use as the lookup table.

  lutCh The digitizer to which *ilut* applies. Must be one of:

  cc8BitInputLutProp::eOnly
  cc8BitInputLutProp::eEven
  cc8BitInputLutProp::eOdd

- `const c_UInt8* inputLut(LutChannel lut=eOnly) const;`

  Returns the 8-bit input lookup table for the specified digitizer.

Constants

default8BitInputLut

`static const c_UInt8 default8BitInputLut[256];`

The default input lookup table is a positive linear mapping in the range 15 to 240. Values below 15 are mapped to 15; values above 240 are mapped to 240.

positive8BitLut `static const c_UInt8 positive8BitLut[256];`

This table is a positive linear mapping from 0 to 255.

shifted8BitLut `static const c_UInt8 shifted8BitLut[256];`

This table is a positive linear mapping with values in the range 0 to 225 shifted to 15 to 240. Values in the range 226 to 255 are clamped to 240. Use this LUT instead of the **default8BitInputLut** when you are using an 8-bit desktop for display and you want to preserve as much information as possible in the low end of the pixel values.

cc8BitInputLutProp

ccAcqFailure

```
#include <ch_cvl/acqfail.h>

class ccAcqFailure;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class returns information about why an acquisition failed.

Constructors/Destructors

ccAcqFailure

```
ccAcqFailure();
~ccAcqFailure();
```

- `ccAcqFailure();`

  Creates an instance of this class. The initial state of this class is successful (no failures).

- `~ccAcqFailure();`

  Destructor.

Operators

bool

```
operator bool();
```

Casting this class to **bool** returns true if the acquisition failed or false otherwise.

Public Member Functions

isMissed

```
bool isMissed();

void isMissed(bool missedState);
```

- `bool isMissed() const;`

  Returns true if the acquisition failed because of a missed trigger. Otherwise returns false. A missed trigger is a hardware trigger which failed to cause an acquisition to occur.

  In general, this failure is an indication that triggers are occurring faster than the acquisition system can process them.

  If you try to start an acquisition on a **ccAcqFifo** object that already has the maximum number of outstanding acquisitions, (*ccAcqFifo::kMaxOutstanding*) the acquisition will fail. When you eventually try to complete the acquisition, the resulting **ccAcqFailure** will be **isMissed()**.

  In semi-trigger mode, you must call **ccAcqFifo::start()** for each acquire. If a trigger occurs and start was not called, the trigger will cause an **isMissed()** error.

  Notes

  On a CVM4 (Cognex CVC-1000 camera), **ccAcqFailure** may report either **isOverrun()** or **isMissed()**.

  If a master FIFO fails because of **isMissed()**, its slaves never receive notification of the missed trigger. To the slave FIFOs, it is as if the triggers never occurred.

- `void isMissed(bool missedState);`

  Sets the missed state of the acquisition.

  Parameters

  missedState The new missed state.

isOverrun

```
bool isOverrun() const;
void isOverrun(bool overrun);
```

- `bool isOverrun() const;`

  Returns true if the acquisition failed because a trigger was received, but an image could not be acquired. Otherwise returns false.

  This error occurs when a trigger caused image integration by the camera, but the image could not be acquired from the camera. Generally, **isOverrun()** errors occur when triggers are arriving too frequently.

  If a master FIFO fails because of an **isOverrun()** condition, the slave FIFOs may successfully complete their acquisitions.

  Notes

  This function always returns false on a Cognex MVS-8120 with CVM1 or CVM6.

  On a CVM4 (Cognex CVC-1000 camera), **ccAcqFailure** may report either **isOverrun()** or **isMissed()**.

- `void isOverrun(bool overrun);`

  Sets the overrun property of the acquisition.

  Parameters

  overrun The overrun state.

isAbnormal

```
bool isAbnormal();
void isAbnormal(bool abnormalState);
```

- `bool isAbnormal();]`

  Returns true if the acquisition failed because a trigger was received, but an image could not be acquired. Otherwise returns false.

  This error occurs when a trigger caused image integration by the camera, but the image could not be acquired from the camera. Generally, **isAbnormal()** errors are caused by factors unrelated to trigger rate, such as PCI bus saturation, blown camera fuse, overheating, and so on. If you are experiencing **isAbnormal()** errors, consult the release notes and hardware-specific documentation to determine the possible causes of the errors.

  For the 8100L, **isAbnormal()** can return true if the camera is not connected, or if the hardware is dropping pixel data because of excessive PCI bus latencies.

Notes

This failure should never happen in a properly functioning system. Some examples of abnormal failures are a blown camera fuse or a temperature alert that indicates that the frame grabber is overheating.

- `void isAbnormal(bool abnormalState);`

Sets the abnormal state of the acquisition.

Parameters

abnormalState The new abnormal state.

isIncomplete

```
bool isIncomplete() const;
void isIncomplete(bool incompleteState);
```

- `bool isIncomplete() const;`

Returns true if the acquisition failed because the **ccAcqFifo::baseComplete()** *maxWait* interval expired before the acquisition entered the complete state (**ccAcqFifo::isComplete()** returns true).

Notes

Although using *maxWait* as a timeout timer is one possible design choice, the preferred approach is to leave *maxWait* set to the default (HUGE_VAL) and the use **ccAcqFifo::isComplete()** to determine when an acquisition is completed. This will avoid **isIncomplete()** errors entirely.

- `void isIncomplete(bool incompleteState);`

Sets the incomplete state of the acquisition.

Parameters

incompleteState The new incomplete state.

isTooFastEncoder

```
bool isTooFastEncoder() const;
void isTooFastEncoder(bool tooFastState);
```

- `bool isTooFastEncoder() const;`

  Returns true if the acquisition failed because of a line overrun situation. This situation is the result of one or more of the following conditions:

  - The line rate being dictated by the position counter is exceeding the output line rate of the camera.
  - The value given to **ccEncoderProp::stepsPerLine()** is too small given the rate at which the position counter increments or decrements/.
  - The value given to **ccExposureProp::exposure()** maps to a line exposure interval that exceeds the length of a line as specified by the position counter and **ccEncoderProp::stepsPerLine()**. This condition applies only when using cameras that support a variable line exposure interval.

  Notes

  This error occurs only with line scan cameras.

- `void isTooFastEncoder(bool tooFastState);`

  Sets the “too fast” state of the acquisition.

  Parameters

  tooFastState The new “too fast” state.

isInvalidRoi

```
bool isInvalidRoi() const;
void isInvalidRoi(bool invalid);
```

`bool isInvalidRoi() const;`

Returns true if an acquisition could not be performed because the region of interest property was invalid (see **ccRoiProp** on page 2763). Typically, a region of interest that is too large will generate this error.

Notes

Some cameras, such as line scan cameras, are implemented with a direct relationship between the ROI property and the image acquired. In these cases, it is possible to specify a ROI outside of the physical bounds of the camera and frame grabber combination.

```
void isInvalidRoi(bool invalid);
```

Sets the invalid region of interest property of the acquisition.

Parameters

invalid The invalid state.

abnormalErrorCode

```
c_Int32 abnormalErrorCode() const;
void abnormalErrorCode(c_Int32);
```

For Cognex use only. In some circumstances, technical support may ask you to use this function to gather more data to help them locate the problem.

- `c_Int32 abnormalErrorCode() const;`

  Returns error information specific to a particular hardware platform thrown by acquisition hardware because of some unusual condition.

- `void abnormalErrorCode(c_Int32);`

  Sets an abnormal error code.

  Parameters

  c_Int32 The error code.

isTimingError

```
bool isTimingError() const;
void isTimingError(bool flag);
```

- `bool isTimingError() const;`

  Returns true if there is a grievous video timing error that prevented the acquisition from occurring. This is commonly caused by attempting to acquire when the camera is not plugged in, or when the video format is not appropriate for the camera.

Notes

This error will occur only for mismatches that prevent the acquire hardware from working at all. Other mismatches may cause corrupted images but not this error.

- `void isTimingError(bool);`

  Sets the timing error flag.

  Parameters

  flag The timing error flag.

Deprecated Members

isOtherFifoError

```
bool isOtherFifoError() const;
void isOtherFifoError(bool error);
```

This function has been deprecated and is maintained for backward compatibility only. It should not be used in new applications.

- `bool isOtherFifoError() const;`

  Returns true if the acquisition failed because of a failure on a master or slave acquisition FIFO.

- `void isOtherFifoError(bool error);`

  Sets the other error property of the acquisition.

  Parameters

  error The error state.

isTimeout

```
bool isTimeout();
void isTimeout(bool timeoutState);
```

This function has been deprecated and is maintained for backward compatibility only. It should not be used in new applications.

- `bool isTimeout();`

  Returns true if the acquisition failed because the timeout period elapsed before the acquisition system was able to obtain the required resources. Returns false otherwise. Timeout errors can occur when the required hardware is being used by another FIFO.

  Notes

  This error is never returned in combination with **isMissed()** or **isOverrun()**.

- `void isTimeout(bool timeoutState);`

  Sets the timeout state of the acquisition.

  Parameters

  timeoutState The new timeout state.

ccAcqFifo

```
#include <ch_cvl/acqbase.h>

class ccAcqFifo : public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

You create **ccAcqFifo** objects by calling **ccStdVideoFormat::newAcqFifoEx()** which returns various instantiations derived from this class. You use **ccAcqFifo** methods to acquire images for all concrete fifo types.

Notes

Using **ccAcqFifo::complete()** in derived classes to handle acquired images is now replaced by **ccAcqFifo::completeAcq()**. The older method is retained for backward compatibility, but you should use **ccAcqFifo::completeAcq()** for all new code.

Constructors/Destructors

ccAcqFifo

```
virtual ~ccAcqFifo();

ccAcqFifo(ccAcqFifo&);
```

- `virtual ~ccAcqFifo();`

  Destroys the FIFO and cancels all outstanding acquisitions on it.

- `ccAcqFifo(ccAcqFifo&);`

  Copy constructor.

Enumerations

ceStartReqStatus

```
enum ceStartReqStatus;
```

These constants provide additional information about the state of an acquisition request after a call to **baseComplete()** or **complete()** in the derived classes.

Value	Meaning
ckStartReqWasServiced	• A new call to **start()** is needed to get the next image. • The returned *appTag* is valid. • The **complete()** call is in sync with the other FIFOs of the master-slave sequence.
ckStartReqNotServiced	• A new call to **start()** is not needed to get the next image. • The returned *appTag* is not valid. • The **complete()** call is out of sync with the other FIFOs of the master-slave sequence.

enum

```
enum {kMaxOutstanding = 32};
```

The maximum number of outstanding acquisitions supported per FIFO. Attempting to have more than this number of acquisitions outstanding at one time causes either **start()** to return false (if the trigger model specifies *ckRequestAcquireOrFail* as its **startAction()**) or a *ccAcqFailure::isMissed* failure otherwise.

Public Member Functions

start

```
bool start(c_UInt32 appTag = 0);
```

Requests that an acquisition be started as soon as possible. The request is added to the FIFO of outstanding acquisitions. A copy of the current property values is stored with the request, so that subsequent changes to this FIFO's properties have no effect on the outstanding acquisitions.

Returns true to indicate the start request was successful. A return of false means the resources were not available and the requested acquisition will not take place.

appTag is an arbitrary value supplied by the application for identifying this acquisition request. The value is returned by the matching **complete()** or **baseComplete()** invocation. The value is not interpreted by **ccAcqFifo**. Typically, the *appTag* is used to facilitate FIFO order integrity verification.

Parameters

appTag — An application defined value that identifies this acquisition request.

Notes

The exact behavior of **start()** is controlled by **ccTriggerModel::startAction()**. See **ceStartAction** on page 3148.

See **prepare()** for a description of what it means to start an acquisition "as soon as possible."

triggerEnabled() must be true for acquisitions to proceed.

Throws

ccAcqFifo::StartNotAllowed

The selected trigger model does not allow **start()** to be invoked. See **ceStartAction** on page 3148.

baseComplete

```
cc_PelBuffer* baseComplete (
  const ccAcqFifo::CompleteArgs& args);

cc_PelBuffer* baseComplete (
  ccAcqFailure* failure=0,
  c_UInt32* appTag = 0,
  bool makeLocal = true,
  double maxWait = HUGE_VAL,
  bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Notes

Using **baseComplete()** or **complete()** in derived classes to handle acquired images is now replaced by **ccAcqFifo::completeAcq()**. The older method is retained for backward compatibility, but you should use **ccAcqFifo::completeAcq()** for all new code.

Classes derived from this class override this function to return a more specific **ccPelBuffer<P>** type. Derived classes typically also supply a function, **complete()**, which is similar to **baseComplete()** but which returns a **ccPelBuffer** object rather than a pointer to one.

The returned pel buffer is a window onto the valid pixels of a pelroot. The underlying pelroot may be larger than this window. For example, the width of the underlying pelroot is often larger to accommodate DMA transfer alignment restrictions imposed by each frame grabber. In this case, the returned pel buffer is a window onto the valid pixels, and the underlying pelroot is larger and contains invalid pixel data. Avoid resizing or repositioning the pel buffer to include the invalid pixel data, since there is no guarantee on the contents of this memory.

The first **baseComplete()** overload takes a **ccAcqFifo::CompleteArgs** object to specify its parameters. Using **CompleteArgs** allows you to specify any parameter in any order without having to specify the preceding parameters. For more information on the **baseComplete()** parameters see **CompleteArgs** on page 3881.

Using the second **baseComplete()** overload you specify individual parameters as call arguments. This syntax has the following limitations, compared to the first overload:

1. Arguments must be specified in the given order, even when not making use of a particular argument. For example, if you wish only to change the *maxWait* timeout value, you must also specify the three preceding parameters.
2. You cannot specify the use of a **ccAcquireInfo** object to contain the results of the image acquisition. The **ccAcquireInfo::triggerNum()** feature does not have an analogous argument in the old form, and thus cannot be used with the old form.

The recommended way to examine or test the results of an image acquisition is to specify a **ccAcquireInfo** object to contain the results. This object is specified as a **CompleteArgs** parameter, as shown in the second example on page 222. For more information on **ccAcquireInfo** objects, see **ccAcquireInfo** on page 261.

- ```
  cc_PelBuffer* baseComplete (
    const ccAcqFifo::CompleteArgs& args);
  ```

  Specify parameters in a **CompleteArgs** object. See **CompleteArgs** on page 3881.

  Returns the image of the oldest completed outstanding image acquisition in the FIFO, and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one.

  Returns null if the acquisition failed or if the **ccAcqFifo::CompleteArgs().maxWait()** period has elapsed. In this case, the **ccAcquireInfo** object you specified with **ccAcqFifo::CompleteArgs** contains the reason the acquisition failed.

  Parameters

  args A completion arguments object.

  Throws

  ccPel::BadWindow

  You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.
  ```

Example

This code fragment gets an image from an outstanding acquisition:

```
auto_ptr<cc_PelBuffer> pb(fifo->baseComplete());
if (pb.get())
  processImage(*pb);
```

This code fragment determines how the acquisition failed:

```
ccAcquireInfo info;
ccPtrHandle<cc_PelBuffer> pb = fifo->baseComplete
  (ccAcqFifo::CompleteArgs().acquireInfo(&info));
if (pb)
  processImage(*pb);
else if (info.failure().isTimeout())
  // Clean up after the timeout
else
  // Report or log an unexpected acquisition failure
```

-
```
cc_PelBuffer* baseComplete (
  ccAcqFailure* failure=0,
  c_UInt32* appTag = 0,
  bool makeLocal = true,
  double maxWait = HUGE_VAL,
  bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Specify the parameters as arguments.

Parameters

failure	See the description of **ccAcquireInfo::**failure() on page 262.
appTag	See the description of **ccAcquireInfo::**appTag() on page 262.
makeLocal	See the description of **ccAcqFifo::CompleteArgs().makeLocal()** on page 3883.
maxWait	See the description of **ccAcqFifo::CompleteArgs().maxWait()** on page 3884.
autoStart	Deprecated parameter. See the description of **ccAcqFifo::CompleteArgs().autoStart()** on page 3885.
startReqStatus	See the description of **ccAcqFifo::CompleteArgs().startReqStatus()** on page 3884.

completeAcq

```
virtual ccAcqImagePtrh completeAcq(
  const CompleteArgs& args = CompleteArgs()) = 0;
```

Gets the image of the oldest outstanding acquisition. This call blocks if there is no outstanding completed acquisition.

The function returns the acquired image. If the acquisition failed or the *maxWait* period elapsed the image will be unbound.

Parameters

args A completion arguments object.

isValid

```
bool isValid(ccAcqProblem* problem = 0) const;
```

Returns true if the FIFO is configured properly and is able to perform an image acquisition.

Parameters

problem If this function returns false, this optional argument contains the reason that the FIFO was not configured properly.

Notes

isValid() performs a more comprehensive check of a FIFO's master/slave configuration than the **ccTriggerProp::triggerMaster()** and **ccTriggerProp::couldSlaveTo()** methods. If a master/slave configuration does not perform as expected, use **isValid()** to check whether the FIFOs are configured properly. If **isValid()** returns false, use **ccAcqProblem::hasInvalidMaster()** or **ccAcqProblem::hasInvalidSlave()** to diagnose the problem.

isIdle

```
bool isIdle() const;
```

Returns true if there are no outstanding acquisitions on the FIFO.

isWaiting

```
bool isWaiting() const;
```

Becomes true when the FIFO begins waiting for **start()**, and returns to false when hardware resources are allocated and all setup delays are finished.

isAcquiring

```
bool isAcquiring() const;
```

Returns true if the oldest outstanding acquisition is waiting for a trigger signal or is acquiring an image. For master/slave acquisitions, it becomes true only after the master and all slaves are ready for acquisition.

isMovable `bool isMovable() const;`

Returns true if the oldest outstanding acquisition has progressed to the point where the camera's field of view can be changed without affecting the acquired image. For example, for strobed and shuttered acquisitions, this means that the strobe or the shutter has been fired.

When an acquisition enters this state, the acquisition software invokes the move-part callback function that you can specify with the move-part callback property. See **ccMovePartCallbackProp** on page 1899.

isComplete `bool isComplete() const;`

Returns true if the oldest outstanding acquisition has completed, perhaps unsuccessfully. The next invocation of **baseComplete()** returns the image if the acquisition was successful or a null pointer if it was not.

When an acquisition enters this state, the acquisition software invokes the acquisition complete callback function that you can specify with the acquisition complete callback property. See **ccCompleteCallbackProp** on page 1049.

pendingAcqs `c_Int32 pendingAcqs() const;`

Returns the number of acquisitions in the *pending* state. This is the number of acquisitions requested by **start()** for which acquisition has not started. To achieve high frame rate acquisition in manual trigger mode, the FIFO must always have one or more pending acquisitions.

completedAcqs `c_Int32 completedAcqs() const;`

Returns the number of acquisitions in the *completed* state. This is the number of completed acquisitions for which the user has not called **complete()**. This is similar to **isComplete()**, except it indicates if more than one completed acquisition is available. If images are being acquired faster than they are being consumed, the number returned by **completedAcqs()** will begin increasing. If left unchecked, this will eventually result in *isMissed* or other errors. **completedAcqs()** can be used to control production line speed to maximize the inspection rate without generating errors.

availableAcqs `c_Int32 availableAcqs() const;`

Returns the number of acquisitions in the *available* state. This method returns an upper bound on the number of additional acquisitions that may be added to the FIFO before errors occur. In manual or semi-automatic trigger mode, calling **start()** decreases the

number of available acquisitions. In auto trigger mode, each trigger decreases the number of available acquisitions. In all trigger modes, calling **complete()** and releasing pel buffers increases available acquisitions.

Notes

The number returned by **availableAcqs()** is an upper bound and not a guaranteed minimum. It is intended to be used to detect resource leaks during debugging.

The sum of (pending + completed + available) is not a constant. Acquisitions in progress are neither pending nor completed.

Calling **start()** does not always decrease **availableAcqs()** by one.

Actions of other FIFOs may affect **availableAcqs()**.

pendingAcqs() and **completedAcqs()** function the same on all platforms and CVL revisions. **availableAcqs()** may behave differently between platforms and CVL revisions. For example, the way available image memory affects **availableAcqs()** can vary.

prepare

```
public: bool prepare(ccCvlString& result,
  double maxWait = 0.0);

bool prepare(double maxWait = HUGE_VAL);
```

This function prepares the video subsystem as needed for this FIFO, waiting up to *maxWait* seconds before returning. It returns true if successful and false if not successful. If **prepare()** returns false, do not attempt an acquisition with this FIFO, as it will fail and likely hang.

Calling **prepare()** does not eliminate or reduce the setup time, but allows the same setup time to occur prior to calling **start()**. This is useful if an acquisition cannot be started immediately, but you want to get the hardware ready now. If an acquisition can start

immediately, there is no performance benefit to calling **prepare()** before **start()**. In this case, the primary use of **prepare()** is to detect hardware setup problems, such as a missing camera.

For an acquisition to take place, the FIFO requires that a certain subset of the video subsystem be in the proper state. For example, a FIFO created to acquire images in RS-170 640x480 format, requires that the video subsystem switch to RS-170 640x480 timing. If the video subsystem section selected by this FIFO is already in this state, the FIFO is considered prepared, and acquisitions can begin immediately.

If the FIFO is not prepared when an acquisition starts, the acquisition software starts preparing the video subsystem automatically and blocks all subsequent acquisitions for that FIFO until the FIFO is prepared or until the timeout limit is reached. Be aware that certain state transitions of the video subsystem take a significant amount of time (up to hundreds of milliseconds) to complete.

The **prepare()** and **isPrepared()** functions let you make sure that an acquisition can start immediately.

Notes

isPrepared() is deprecated..

In a typical application, the acquisition FIFOs are always prepared and all acquisitions start immediately, so most applications do not need to use the **prepare()** function.

Parameters

maxWait	The number of seconds to wait until the video subsystem is prepared before returning. The value *HUGE_VAL* (defined in *<math.h>)* indicates that the system should wait indefinitely. The FIFO should be idle before calling **prepare()**. If it is not idle (starts are queued, or you are in auto trigger mode with triggers enabled), **prepare()** will assume the hardware is already prepared and will return true immediately. When the FIFO is idle there should be no wait, so Cognex recommends you always set *maxWait* to 0.
result	Additional information about what happened during the prepare.

Notes

On most platforms, *maxWait* has a granularity between 1 millisecond and 1 microsecond.

If **ccTriggerProp::triggerEnable()** is false and **start()** calls are queued, **prepare()** will return immediately without preparing the FIFO. If you use **prepare()** you should call **prepare()** and then **start()**, not the reverse.

Calling **prepare()** on a master or slave FIFO prepares all the associated master and slave FIFOs. You could prepare each master and slave FIFO individually, but that would be redundant.

You do not need to call **prepare()** when using automatic triggering. The acquisition software performs hardware preparation for you when **triggerEnable()** is true.

Additional information about what happened during the prepare is available via the result string.

flush

```
void flush();
```

Discards all outstanding acquisitions, leaving this FIFO in the idle state. If an acquisition is in progress, it is cancelled and discarded. If **ccTriggerProp::triggerEnable()** is true, this function disables triggers before discarding all acquisitions, then re-enables triggers.

properties

```
virtual ccAcqProps& properties()= 0;
```

Returns a reference to the FIFO's properties object.

Notes

The reference that this function returns is not **const** so that you can invoke appropriate **ccAcqProps** member functions to set individual properties. Derived classes typically override this function to return a more specific type derived from **ccAcqProps**.

propertyQuery

```
ccAcqPropertyQuery& propertyQuery();
```

Returns a property query object that you can use to test whether the FIFO supports a particular property. If the property is supported, this function returns a valid property object. The returned object may be used to set property values. Both pointer and reference semantics are supported.

Example

To set the timeout property only if this FIFO supports it, you would write:

```
ccTimeoutProp* timeoutProp = fifo->propertyQuery();
if (timeoutProp)
  timeoutProp->timeout(10.0);
```

Or, using reference semantics, you would write:

```
try
{
  ccTimeoutProp& timeoutProp = fifo->propertyQuery();
  timeoutProp.strobeEnable(true);
}
catch (cmStd bad_cast&)
{
  // The timeout property is not supported.
}
```

---

triggerModel

```
void triggerModel(const ccTriggerModel& model);

void triggerModel(ccTriggerModelPtrh model);

const ccTriggerModel& triggerModel() const;
```

- `void triggerModel(const ccTriggerModel& model);`

  Sets the trigger model for the current FIFO. The default model is **cfManualTrigger()**. See **ccTriggerModel** on page 3145.

  Parameters

  model — A trigger model object, usually created with one of the global functions **cfManualTrigger()**, **cfAutoTrigger()**, **cfSemiTrigger()**, or **cfSlaveTrigger()**.

- `void triggerModel(ccTriggerModelPtrh model);`

  Sets the trigger model for the current FIFO. Cognex recommends that custom trigger models be set using the **ccTriggerModelPtrh** overload. Built-in trigger models (such as **cfManualTrigger**) can only be set using the **ccTriggerModel&** overload.

  Parameters

  model — The name of your custom trigger model object.

- `const ccTriggerModel& triggerModel() const;`

  Returns the currently set trigger model for this FIFO.

  See also **cfManualTrigger()** on page 3733, **cfAutoTrigger()** on page 3533, **cfSemiTrigger()** on page 3853, and **cfSlaveTrigger()** on page 3861.

triggerEnable

```
void triggerEnable(bool enable);

bool triggerEnable() const;
```

- `void triggerEnable(bool enable);`

  Enables or disables triggers. The default is true.

  Invoking **triggerEnable()** for a master camera channel or any of its slaves has the effect of invoking **triggerEnable()** for all. That is, masters and slaves have the same **triggerEnable()** setting.

  Parameters

  enable — In general, true allows acquisitions to happen and false prevents acquisitions from happening, depending on the current trigger model and its state. For more information on each trigger model, see **cfManualTrigger()** on page 3733, **cfAutoTrigger()** on page 3533, **cfSemiTrigger()** on page 3853, and **cfSlaveTrigger()** on page 3861.

- `bool triggerEnable() const;`

  Returns true if triggers are enabled, false otherwise.

movePartInfoCallback

```
void movePartInfoCallback(const ccCallbackAcqInfoPtrh&);
const ccCallbackAcqInfoPtrh& movePartInfoCallback() const;
```

- `void movePartInfoCallback(const ccCallbackAcqInfoPtrh&);`

This function is one of two ways to register a callback class for the movable state:

Method	Features	See
ccAcqFifo:: movePartInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	this section
ccMovePartCallbackProp	Calls your callback function.	*ccMovePartCallbackProp* on page 1899

This function effectively registers a callback function that you want the acquisition engine to call when the acquisition enters the movable state (that is, when **ccAcqFifo::isMovable()** returns true). The movable state is when the camera's field of view can be changed without affecting the acquired image.

An unbound handle means no callback will occur.

This function actually registers, not a function, but a callback class you define. Your callback class contains your callback function, defined as an override of **operator()()** of that class.

A **ccCallbackAcqInfo** class is a special case of a **ccCallback1** class in which an instance of **ccAcquireInfo** is passed as an argument. This allows your callback function to use the information about the current state of the acquisition stored in the **ccAcquireInfo** object.

The callback function you write should set flags or semaphores in your application that allow your program to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

In general, an acquisition enters the movable state during the acquisition's next-to-last vertical blank interval. With an RS-170 camera, for example, an acquisition would enter the movable state 17 ms (one field time) before it enters the complete state.

Not all frame grabbers can detect the next-to-last vertical blank interval. When using one of these frame grabbers, the acquisition enters the movable and complete states at the same time. In such cases, the move-part callback is always invoked first, followed immediately by the completion callback.

On the MVS-8100M, MVS-8100C, and MVS-8100C/CPCI (but not the MVS-8100M+ when using CCF-based video formats), you can select between default and optimum timing of the move-part callback function's execution. See **cc8100m::movePartTimingChoice** on page 210.

For a further discussion, see *Using Callback Functions* in the *Acquiring Images* chapter of the *CVL User's Guide.*

Parameters

ccCallbackAcqInfoPtrh

Pointer handle referencing an instance of a callback class you've defined, that contains the callback function you have defined as an override of **operator()()** for that class.

- `const ccCallbackAcqInfoPtrh& movePartInfoCallback() const;`

  Returns a pointer handle to the registered callback class instance for this acquisition.

completeInfoCallback

```
void completeInfoCallback(const ccCallbackAcqInfoPtrh&);
const ccCallbackAcqInfoPtrh& completeInfoCallback() const;
```

- `void completeInfoCallback(const ccCallbackAcqInfoPtrh&);`

This function is one of two ways to register a callback class for the complete state:

Method	Features	See
ccAcqFifo:: completeInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	this section
ccCompleteCallbackProp	Calls your callback function.	*ccCompleteCallbackProp* on page 1049

This function effectively registers a callback function that you want the acquisition engine to call when the image acquisition is complete. An acquisition enters the complete state (**ccAcqFifo::isComplete()** returns true) when the next invocation of **ccAcqFifo::baseComplete()** would return the acquired image (if the acquisition was successful) or null (if the acquisition was not successful).

An unbound handle means no callback will occur.

This function actually registers, not a function, but a callback class you define. Your callback class contains your callback function, defined as an override of **operator()()** of that class.

A **ccCallbackAcqInfo** class is a special case of a **ccCallback1** class in which an instance of **ccAcquireInfo** is passed as an argument. This allows your callback function to use the information about the current state of the acquisition stored in the **ccAcquireInfo** object.

The callback function you write should set flags or semaphores in your application that allow your program to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

For a further discussion, see *Using Callback Functions* in the *Acquiring Images* chapter of the *CVL User's Guide.*

Parameters

ccCallbackAcqInfoPtrh

Pointer handle referencing an instance of a callback class you've defined, that contains the callback function you have defined as an override of **operator()()** for that class.

- `const ccCallbackAcqInfoPtrh& completeInfoCallback() const;`

  Returns a pointer handle to the registered callback class instance for this acquisition.

overrunInfoCallback

```
void overrunInfoCallback(const ccCallbackAcqInfoPtrh&);
const ccCallbackAcqInfoPtrh& overrunInfoCallback() const;
```

- `void overrunInfoCallback(const ccCallbackAcqInfoPtrh&);`

This function is one of two ways to register a callback class for the overrun state:

Method	Features	See
ccAcqFifo:: overrunInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	this section
ccOverrunCallbackProp	Calls your callback function.	*ccOverrunCallbackProp* on page 2345

This function effectively registers a callback function that you want the acquisition engine to call if an overrun occurs. Generally, overrun errors and **isMissed()** errors occur when triggers occur too fast. See **ccAcqFailure::isOverrun()** and **ccAcqFailure::isMissed()** for more information about the overrun state.

An unbound handle means no callback will occur.

This function actually registers, not a function, but a callback class you define. Your callback class contains your callback function, defined as an override of **operator()()** of that class.

A **ccCallbackAcqInfo** class is a special case of a **ccCallback1** class in which an instance of **ccAcquireInfo** is passed as an argument. This allows your callback function to use the information about the current state of the acquisition stored in the **ccAcquireInfo** object.

The callback function you write should set flags or semaphores in your application that allow your program to handle the overrun condition the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

For a further discussion, see *Using Callback Functions* in the *Acquiring Images* chapter of the *CVL User's Guide*.

Parameters

ccCallbackAcqInfoPtrh

Pointer handle referencing an instance of a callback class you've defined, that contains the callback function you have defined as an override of **operator()()** for that class.

- `const ccCallbackAcqInfoPtrh& overrunInfoCallback() const;`

  Returns a pointer handle to the registered callback class instance for this acquisition.

videoFormat

```
virtual const ccVideoFormat& videoFormat() const = 0;
```

Returns the video format for which this FIFO was constructed.

frameGrabber

```
virtual ccFrameGrabber& frameGrabber() const = 0;
```

Returns the frame grabber for which this FIFO was constructed.

hardwareImagePoolSize

```
c_Int32 hardwareImagePoolSize() const;
void hardwareImagePoolSize(c_Int32 numberOfImages);
```

- `c_Int32 hardwareImagePoolSize() const;`

  Returns the size of the MVS-8600 hardware image pool in number of images.

- `void hardwareImagePoolSize(c_Int32 numberOfImages);`

  Sets the size, in number of images, of the MVS-8600 hardware image pool used the by the acquisition FIFO. A larger image pool size reduces the likelihood of acquisition errors due to system latency or high CPU load. A smaller pool size conserves memory.

  For area scan cameras the amount of memory, in bytes, used for the image pool is:

  videoFormatWidth * videoFormatHeight * bytesPerPixel * numberOfImages.

  For line scan cameras the amount of memory in bytes is:

  videoFormatWidth * imageROIHeight * bytesPerPixel * numberOfImages.

  In most cases, a minimum pool size of 4 images is recommended. In the case of images larger than 32MB, a pool size of 2 or 3 images may be used to conserve memory if the acquisition rate allows..

Parameters

numberOfImages

The number of images to allocate in the hardware pool. The lowest number allowed is 2 images. Specifying zero sets the number of images to **defaultHardwareImagePoolSize()**.

Notes

This function applies only to the MVS-8600 and MVS-8600e frame grabbers.

If there isn't enough memory available to create an image pool of the given size, **prepare()** returns False, **completeAcq()** fails, and **isAbnormal()** is True with error code 2. In this case, try reducing the pool size. If the pool is exhausted at runtime, **completeAcq()** fails, and **isAbnormal()** is True with error code 7. In this case, try increasing the pool size.

Throws

ccAcqFifo::BadParams

numberOfImages is 1 or less than zero,.

defaultHardwareImagePoolSize

```
c_Int32 defaultHardwareImagePoolSize() const;
```

Returns the default number of images available in the MVS-8600 image pool. For line scan acquisition default the number of images depends on the image height which you can adjust with the region of interest (ROI) setting.

Notes

This function applies only to the MVS-8600 and MVS-8600e frame grabbers.

Deprecated Members

isPrepared

```
bool isPrepared() const;
```

Returns true if the resources that the FIFO needs to perform an acquisition are ready. If **isPrepared()** returns true, an acquisition request begins immediately.

This function is deprecated in CVL 6.2 cr10.

Friends

ccDirectDrawSurfacePool
```
friend class ccDirectDrawSurfacePool;
```

cc_FGDisplay
```
friend class cc_FGDisplay;
```

Typedefs

ccAcqFifoPtrh
```
typedef ccPtrHandle<ccAcqFifo> ccAcqFifoPtrh;
```

ccAcqFifoPtrh_const
```
typedef ccPtrHandle_const<ccAcqFifo> ccAcqFifoPtrh_const;
```

ccAcqImage

```
#include <acqimage.h>

class ccAcqImage : public virtual ccRepBase
```

Class Properties

Copyable	No
Derivable	Not Intended
Archiveable	Yes

ccAcqImage holds the information for an acquired image. A **ccAcqImage** object containing the acquired image is returned when you call **ccAcqFifo::completeAcq()**.

This class contains the public member functions you can call to convert the acquired image into various formats. If the requested format is different from the acquired image format, the acquired image is converted to the requested format and the new image is returned. The acquired image is unchanged.

If the requested format is the same as the acquired format, the acquired image is returned directly without copying. Be aware that in this case you are working with the acquired image, not a copy.

When there is an acquisition error, the returned **ccAcqImage** object will be unbound. In this case, calling any of the conversion routines will also result in unbound images. Check whether an image is bound before making conversion calls.

Note Image format conversions are done completely in software and can have a significant impact on system performance. For a one megapixel image, the conversion can take over 100 milliseconds on a 1 GHz processor.

The **ceImageFormat** enum on page 240 lists the possible acquired image formats. All acquired image formats with the exception of *ceImageFormat_Bayer16* can be converted into formats acceptable to CVL by calling conversion routines in this class.

Constructors/Destructors

ccAcqImage

```
ccAcqImage() {}

ccAcqImage(
  const ccPelBuffer<c_UInt8>& buf,
  ceImageFormat formatAcquired);

ccAcqImage(const ccPelBuffer<ccPackedRGB16Pel>& buf);

ccAcqImage(const ccPelBuffer<ccPackedRGB32Pel>& buf);

ccAcqImage(const cc3PlanePelBuffer& buf);

~ccAcqImage() {}
```

Notes

Since **ccAcqImage** objects are returned to you when you call **ccAcqFifo::completeAcq()** there is no need for you to create **ccAcqImage** objects in your program.

Enumerations

ceImageFormat

```
enum ceImageFormat;
```

These values specify all of the possible acquired image formats.

Value	Format	Bit Depth
ceImageFormat_Grey8	Monochrome	8 bits/pixel
ceImageFormat_Grey10	Monochrome	10 bits/pixel
ceImageFormat_Grey12	Monochrome	12 bits/pixel
ceImageFormat_Grey16	Monochrome	16 bits/pixel
ceImageFormat_PackedRGB16	[R,G,B]=[5,6,5]	16 bits/pixel
ceImageFormat_PackedRGB32	[a,R,G,B]=[8,8,8,8]	32 bits/pixel
ceImageFormat_Bayer8	Bayer8	8 bits/pixel
ceImageFormat_Bayer16	Bayer16	16 bits/pixel
ceImageFormat_UYYVYY	YUV 4:1:1	12 bits/pixel (average)
ceImageFormat_UYVY	YUV 4:2:2	16 bits/pixel (average)

Value	Format	Bit Depth
ceImageFormat_UYV	YUV 4:4:4	24 bits/pixel
ceImageFormat_PackedRGB24	[R,G,B]=[8,8,8]	24 bits/pixel
ceImageFormat_PackedRGB48	[R,G,B]=[16,16,16]	48 bits/pixel
ceImageFormat_PlanarRGB8	R=G=B=8 bits	8 bits/pixel * 3
ceImageFormat_PackedBGR24	[B,G,R]=[8,8,8]	24 bits/pixel
ceImageFormat_Grey16BigEndian	Monochrome	16 bits/pixel
ceImageFormat_Grey14	Monochrome	14 bits/pixel
ceImageFormat_Grey10_Packed	Monochrome	10 bits/pixel (2 pixels/3 bytes)
ceImageFormat_Grey12_Packed	Monochrome	14 bits/pixel (2 pixels/3 bytes)
ceImageFormat_Grey8_2Byte	Monochrome	14 bits/pixel(1pixel/2 bytes)
ceImageFormat_Unknown	Unknown video format	

Public Member Functions

getGrey8PelBuffer

```
ccPelBuffer<c_UInt8> getGrey8PelBuffer();
```

Return the acquired image in *ceImageFormat_Grey8* format. If the acquired image format is not *ceImageFormat_Grey8* the acquired image is converted to *ceImageFormat_Grey8* and the new image is returned. The acquired image is unchanged.

If the acquired image format is *ceImageFormat_Grey8,* the acquired image is returned directly without copying. Be aware that in this case you are working with the acquired image, not a copy.

Throws

ccAcqImage::NotSupported

The conversion of the acquired image format to *ceImageFormat_Grey8* is not supported.

getGrey16PelBuffer

```
ccPelBuffer<c_UInt16> getGrey16PelBuffer();
```

Return the acquired image in *ceImageFormat_Grey16* format. If the acquired image format is not *ceImageFormat_Grey16* the acquired image is converted to *ceImageFormat_Grey16* and the new image is returned. The acquired image is unchanged.

If the acquired image format is *ceImageFormat_Grey16,* the acquired image is returned directly without copying. Be aware that in this case you are working with the acquired image, not a copy.

Throws

ccAcqImage::NotSupported
The conversion of the acquired image format to *ceImageFormat_Grey16* is not supported.

getPackedRGB16PelBuffer

```
ccPelBuffer<ccPackedRGB16Pel> getPackedRGB16PelBuffer();
```

Return the acquired image in *ceImageFormat_PackedRGB16* format. If the acquired image format is not *ceImageFormat_PackedRGB16* the acquired image is converted to *ceImageFormat_PackedRGB16* and the new image is returned. The acquired image is unchanged.

If the acquired image format is *ceImageFormat_PackedRGB16,* the acquired image is returned directly without copying. Be aware that in this case you are working with the acquired image, not a copy.

Throws

ccAcqImage::NotSupported
The conversion of the acquired image format to *ceImageFormat_PackedRGB16* is not supported.

getPackedRGB32PelBuffer

```
ccPelBuffer<ccPackedRGB32Pel> getPackedRGB32PelBuffer();
```

Return the acquired image in *ceImageFormat_PackedRGB32* format. If the acquired image format is not *ceImageFormat_PackedRGB32* the acquired image is converted to *ceImageFormat_PackedRGB32* and the new image is returned. The acquired image is unchanged.

If the acquired image format is *ceImageFormat_PackedRGB32,* the acquired image is returned directly without copying. Be aware that in this case you are working with the acquired image, not a copy.

Throws

ccAcqImage::NotSupported

The conversion of the acquired image format to *ceImageFormat_PackedRGB32* is not supported.

get3PlanePelBuffer

```
cc3PlanePelBuffer get3PlanePelBuffer();
```

Return the acquired image in *ceImageFormat_PlanarRGB8* format. If the acquired image format is not *ceImageFormat_PlanarRGB8* the acquired image is converted to *ceImageFormat_PlanarRGB8* and the new image is returned. The acquired image is unchanged.

If the acquired image format is *ceImageFormat_PlanarRGB8,* the acquired image is returned directly without copying. Be aware that in this case you are working with the acquired image, not a copy.

Throws

ccAcqImage::NotSupported

The conversion of the acquired image format to *ceImageFormat_PlanarRGB8* is not supported.

isBound

```
bool isBound() const;
```

Return true if the image is bound to a root image. Return false otherwise.

isConversionSupported

```
bool isConversionSupported(
  ceImageFormat formatConverted) const;
```

Return true if the *formatConverted* conversion is supported. Return false otherwise.

Parameters

formatConverted

The format to test.

isColor

```
bool isColor() const;
```

Returns true if the image was acquired using a color video format. Returns false otherwise.

width

```
c_Int32 width() const;
```

Return the width of the acquired image in pixels (if the image is bound).

height `c_Int32 height() const;`

Return the height of the acquired image in rows (if the image is bound).

alignModulus `c_Int32 alignModulus() const;`

Return the row alignment of the acquired image in bytes (if the image is bound).

rowUpdate `c_Int32 rowUpdate() const;`

Return the row update of the acquired image in bytes (if the image is bound).

formatAcquired `ceImageFormat formatAcquired() const;`

Return the format of the acquired image (if the image is bound).

intensityBits `c_Int32 intensityBits() const;`

Returns the number of bits of intensity data per pixel if the image is bound or 8 if the image is unbound.

For color images, this function indicates the number of bits per pixel per color. That is, if the image contains 8-bits each of red, green, and blue data, this function returns 8 even though the total number of data bits per pixel is 24.

Notes

For *ceImageFormat_PackedRGB16* images, this function returns 8, rather than the theoretically correct value of 5.33333...

Static Functions

isColor `static bool isColor(ceImageFormat imgfmt);`

Returns true if the supplied image format is a color image.

Typedefs

ccAcqImagePtrh `typedef ccPtrHandle<ccAcqImage> ccAcqImagePtrh;`

ccAcqProblem

```
#include <ch_cvl/acqprob.h>

class ccAcqProblem;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains member functions that help you identify why a FIFO is improperly configured. You should use these functions when **ccAcqFifo::isValid()** returns false. For more information about valid FIFOs and using **ccAcqProblem**, see the *CVL User's Guide* and the *acqprob.cpp* sample program.

Attempting to acquire from an invalid FIFO is not recommended, and can result in image corruption and may hang your program. However, this is a recommendation and is not strictly enforced.

Note that member functions in this class attempt to identify a set of common problems and may not identify every possible problem. You may encounter problems in your application that **ccAcqProblem** does not specifically identify.

Constructors/Destructors

ccAcqProblem

```
ccAcqProblem();

ccAcqProblem(const ccAcqProblem&);

virtual ~ccAcqProblem();
```

- `ccAcqProblem();`

  Default constructor.

- `ccAcqProblem(const ccAcqProblem&);`

  Copy constructor.

- `virtual ~ccAcqProblem();`

  Destructor.

Operators

bool `operator bool() cons;`

Casting this class to **bool** returns true if the acquisition is configured improperly. It returns false otherwise.

operator= `ccAcqProblem& operator=(const ccAcqProblem& prob);`

Assignment operator. Assigns the value of *prob* to this object.

Parameters

prob The value to assign.

Public Member Functions

slavePortInvalid

```
bool slavePortInvalid() const;
```

Returns true if the camera port settings are invalid. While the individual camera port settings for each FIFO may be valid, two FIFOs may be erroneously assigned to the same camera. Another possibility is that the master/slave arrangement may violate a platform-specific requirement. For example, a platform may require the slave FIFO to be on the next higher port from the master.

slaveNotSlaveTrigger

```
bool slaveNotSlaveTrigger() const;
```

Returns true if a FIFO has been assigned a trigger master but not assigned a slave trigger model.

slaveTriggerNoMaster

```
bool slaveTriggerNoMaster() const;
```

Returns true if the FIFO has a slave trigger model but has not been assigned a trigger master.

masterIsSlaveTrigger

```
bool masterIsSlaveTrigger() const;
```

Returns true if trigger master is assigned the slave trigger model. A trigger master cannot be slaved to another FIFO.

mismatchedExposures

```
bool mismatchedExposures() const;
```

Returns true if the master and slaves are using different exposure settings. The exposures for all master/slaved FIFOs must be set to identical values to ensure the cameras stay synchronized for shared strobe applications.

This problem can be ignored when using the CVC-1000 and non-strobed acquisitions.

mismatchedFormats

```
bool mismatchedFormats() const;
```

Returns true if master and slave FIFOs are using different camera formats. In most cases you application will hang when you attempt to acquire images with this condition.

It may be possible to master/slave compatible cameras, for example two different RS-170 type cameras that have identical CTIs. This can occur when individual cameras are replaced with a later model or different brand. However *any* differences in video timing will cause problems in master/slave applications.

invalidCameraPort

```
bool invalidCameraPort() const;
```

Returns true if the camera port setting is outside the range of 0 - **numCameraPort()**-1

invalidAuxLightPort

```
bool invalidAuxLightPort() const;
```

Returns true if lighting is enabled and the aux light port setting is outside the range of 0 - 4, or if it is in conflict with the camera port setting. If an acquire is attempted the aux light will not work.

tooFewPorts

```
bool tooFewPorts() const;
```

Returns true if, in a master/slave configuration, there are not enough camera ports to support the number of taps needed for all the cameras.

badSyncModel

```
bool badSyncModel() const;
```

Returns true if, in a master/slave configuration, a slave is set to an improper sync model.

Deprecated Members

The following functions are deprecated and are maintained for backward compatibility only. These functions should not be used in new applications.

hasInvalidSlave

```
bool hasInvalidSlave() const;
```

Returns true when the FIFO is invalid due to some master/slave problem. Other methods in this class provide information regarding why the master/slave setup is invalid.

This problem can occur due to limitations or constraints of the frame grabber associated with selecting master and slave **ccAcqFifo**s. Check your frame grabber-specific documentation for constraints regarding **ccAcqFifo**s. For example, check the camera port intended for master/slave acquisitions.

hasInvalidMaster

```
bool hasInvalidMaster() const;
```

Returns true when the FIFO is invalid due to some master/slave problem. Other methods in this class provide information regarding why the master/slave setup is invalid.

This problem can occur due to limitations or constraints of the frame grabber associated with selecting master and slave **ccAcqFifo**s. Check your frame grabber-specific documentation for constraints regarding **ccAcqFifo**s. For example, check the camera port intended for master/slave acquisitions.

ccAcqPropertyQuery

```
#include <ch_cvl/prop.h>

class ccAcqPropertyQuery;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class is used with **ccAcqFifo::propertyQuery()** to determine whether a FIFO supports a particular property. When cast implicitly or explicitly to any of the property classes, the resulting cast returns a pointer or a reference to the property class. If the property is not supported, it returns a null pointer, or, in the case of reference semantics, throws *std::bad_cast*.

For example, to set the timeout property only if a FIFO supports it, you would write:

```
ccContrastBrightnessProp* contrastBrightnessProp =
                                          fifo->propertyQuery();
if (contrastBrightnessProp)
    contrastBrightnessProp->contrastBrightness(0.5, 0.5);
```

Or, using reference semantics, you would write:

```
try
{
    ccContrastBrightnessProp& contrastBrightnessProp =
                                          fifo->propertyQuery();
    contrastBrightnessProp.contrastBrightness(0.5, 0.5);
}
catch (cmStd bad_cast&)
{
    // The contrast/brightness property is not supported.
}
```

The constructor and destructor for this class are protected.

Operators

All of the cast operators for the **ccAcqPropertyQuery** class are public.

The cast-to-pointer operators all return a pointer to a property object, if the FIFO supports that property. If not, they return a null pointer.

The cast-to-reference operators all return a reference to a property object, if the FIFO supports that property. If not, they throw an *std::bad_cast* exception. Table 1 lists the type-safe property cast operators and their signatures.

Cast Operator	Signature
operator ccTriggerProp* ()	`virtual operator ccTriggerProp* () const;`
operator ccTriggerProp& ()	`virtual operator ccTriggerProp& () const;`
operator ccTriggerFilterProp* ()	`virtual operator ccTriggerFilterProp* () const;`
operator ccTriggerFilterProp& ()	`virtual operator ccTriggerFilterProp& () const;`
operator ccStrobeProp* ()	`virtual operator ccStrobeProp* () const;`
operator ccStrobeProp& ()	`virtual operator ccStrobeProp& () const;`
operator ccStrobeDelayProp* ()	`virtual operator ccStrobeDelayProp* () const;`
operator ccStrobeDelayProp& ()	`virtual operator ccStrobeDelayProp& () const;`
operator ccExposureProp* ()	`virtual operator ccExposureProp* () const;`
operator ccExposureProp& ()	`virtual operator ccExposureProp& () const;`
operator ccTimeoutProp* ()	`virtual operator ccTimeoutProp* () const;`
operator ccTimeoutProp& ()	`virtual operator ccTimeoutProp& () const;`
operator ccCameraPortProp* ()	`virtual operator ccCameraPortProp* () const;`
operator ccCameraPortProp& ()	`virtual operator ccCameraPortProp& () const;`
operator ccMovePartCallbackProp* ()	`virtual operator ccMovePartCallbackProp* () const;`
operator ccMovePartCallbackProp& ()	`virtual operator ccMovePartCallbackProp& () const;`
operator ccOverrunCallbackProp* ()	`virtual operator ccOverrunCallbackProp* () const;`
operator ccOverrunCallbackProp& ()	`virtual operator ccOverrunCallbackProp& () const;`
operator ccCompleteCallbackProp* ()	`virtual operator ccCompleteCallbackProp* () const;`
operator ccCompleteCallbackProp& ()	`virtual operator ccCompleteCallbackProp& () const;`

*Table 1. Cast operators used for property queries with **ccAcqPropertyQuery** class*

Cast Operator	Signature
operator ccPelRootPoolProp* ()	`virtual operator ccPelRootPoolProp* () const;`
operator ccPelRootPoolProp& ()	`virtual operator ccPelRootPoolProp& () const;`
operator ccRoiProp* ()	`virtual operator ccRoiProp* () const;`
operator ccRoiProp& ()	`virtual operator ccRoiProp& () const;`
operator cc8BitInputLutProp* ()	`virtual operator cc8BitInputLutProp* () const;`
operator cc8BitInputLutProp& ()	`virtual operator cc8BitInputLutProp& () const;`
operator ccContrastBrightnessProp* ()	`virtual operator ccContrastBrightnessProp* () const;`
operator ccContrastBrightnessProp& ()	`virtual operator ccContrastBrightnessProp& () const;`
operator ccFirstPelOffsetProp* ()	`virtual operator ccFirstPelOffsetProp* () const;`
operator ccFirstPelOffsetProp& ()	`virtual operator ccFirstPelOffsetProp& () const;`
operator ccEncoderControlProp* ()	`virtual operator ccEncoderControlProp* () const;`
operator ccEncoderControlProp& ()	`virtual operator ccEncoderControlProp& () const;`
operator ccEncoderProp* ()	`virtual operator ccEncoderProp* () const;`
operator ccEncoderProp& ()	`virtual operator ccEncoderProp& () const;`
operator ccDigitalCameraControlProp* ()	`virtual operator ccDigitalCameraControlProp* () const;`
operator ccDigitalCameraControlProp& ()	`virtual operator ccDigitalCameraControlProp& () const;`
operator ccSampleProp* ()	`virtual operator ccSampleProp* () const;`
operator ccSampleProp& ()	`virtual operator ccSampleProp& () const;`
operator ccCustomProp* ()	`virtual operator ccCustomProp* () const;`
operator ccCustomProp& ()	`virtual operator ccCustomProp& () const;`
operator ccGigEVisionTransportProp* ()	`virtual operator ccGigEVisionTransportProp* () const;`
operator ccGigEVisionTransportProp& ()	`virtual operator ccGigEVisionTransportProp& () const;`

*Table 1. Cast operators used for property queries with **ccAcqPropertyQuery** class*

ccAcqPropertyQuery

ccAcqProps

```
#include <ch_cvl/prop.h>

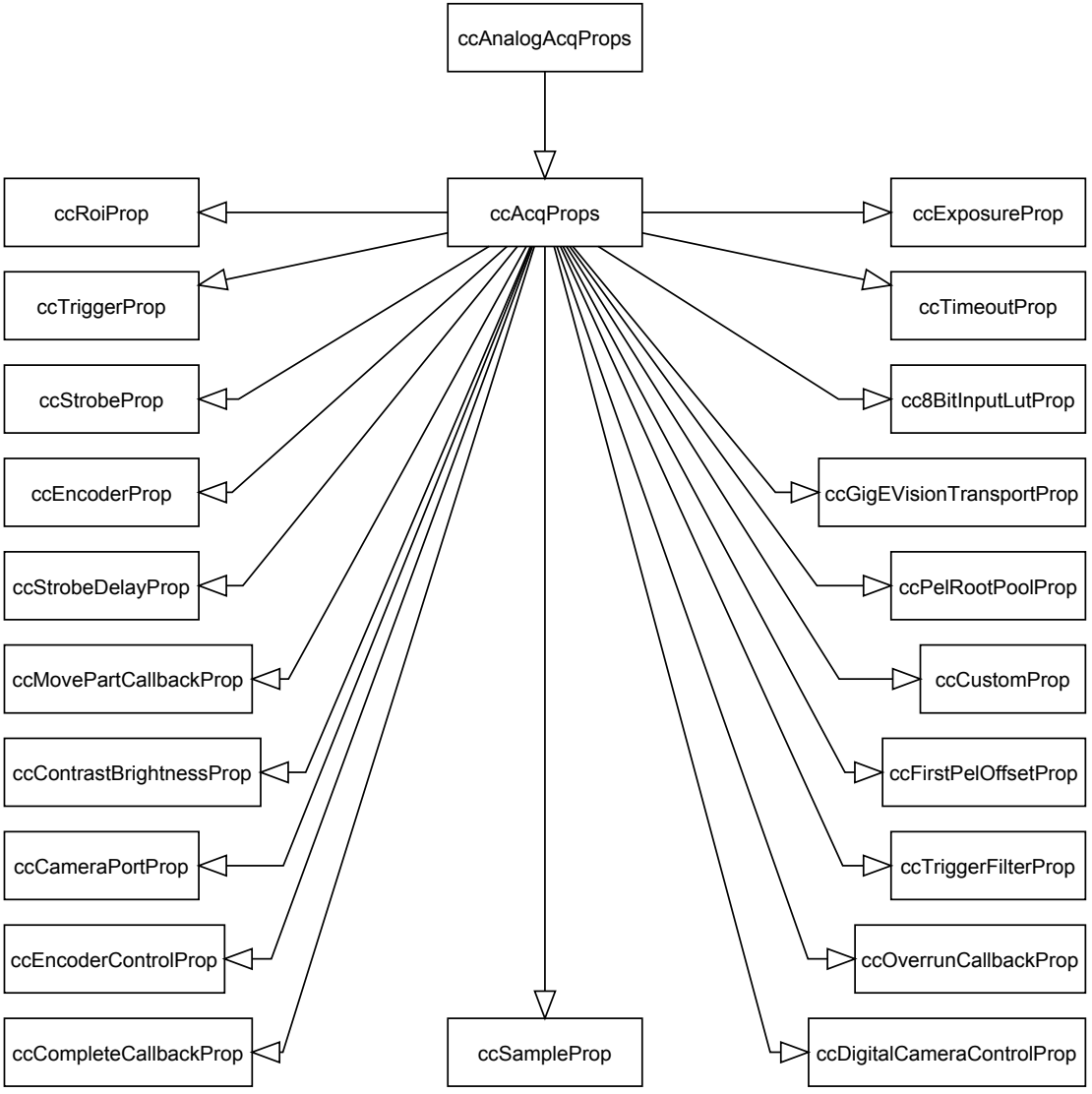
class ccAcqProps :
  public ccTriggerProp,
  public ccTriggerFilterProp,
  public ccStrobeProp,
  public ccStrobeDelayProp,
  public ccExposureProp,
  public ccTimeoutProp,
  public ccCameraPortProp,
  public ccMovePartCallbackProp,
  public ccOverrunCallbackProp,
  public ccCompleteCallbackProp,
  public ccPelRootPoolProp,
  public ccRoiProp,
  public cc8BitInputLutProp,
  public ccContrastBrightnessProp,
  public ccFirstPelOffsetProp,
  public ccEncoderControlProp,
  public ccEncoderProp,
  public ccDigitalCameraControlProp,
  public ccSampleProp,
  public ccCustomProp,
  public ccGigEVisionTransportProp
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This is the base class for all properties that are common to all **ccAcqFifo** types.

ccAcqProps



The above figure shows the **ccAcqProps** class inheritance hierarchy.

Constructors/Destructors

ccAcqProps `ccAcqProps();`

Creates a properties object not associated with any FIFO. All values are default values.

ccAcqProps

ccAcuBarCodeCalibrationResult

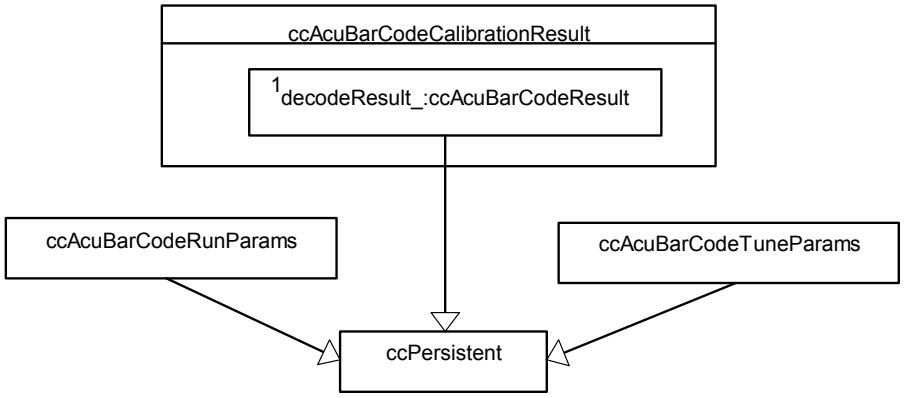
```
#include <ch_cvl/acubar.h>

class ccAcuBarCodeCalibrationResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the results of applying the calibrate operation to the Barcode tool.



Constructors/Destructors

Uses the compiler-generated default constructor, copy constructor, destructor, and assignment operator.

Public Member Functions

isCalibrated

```
bool isCalibrated() const;
```

Returns true if calibration is successful, false otherwise. Calibration is considered successful if the associated decode operation, using the calibrated barcode properties, is successful, that is if the score of the decode operation is greater than or equal to the accept threshold.

symbolType

```
ccAcuBarCodeDefs::Symbology symbolType() const;
```

Returns the type of 1D symbology found. The symbology must have been computed as a result of calibration.

resultWindow

```
ccPelRect resultWindow(bool includeQuietZone) const;
```

Returns an image-coordinate-aligned rectangle that best fits the limits of the barcode.

Parameters

includeQuietZone

If *includeQuietZone* is true, the rectangle returned includes the leading and trailing quiet zones. If *includeQuietZone* is false, the rectangle returned encloses just the barcode and does not include the leading and trailing quiet zones.

Notes

The position of the barcode is the center of this best-fit rectangle.

pitch

```
double pitch() const;
```

Returns the pitch of the barcode in pixels. The pitch of the barcode is the average spacing between consecutive bars.

stringLength

```
c_Int32 stringLength() const;
```

Returns the length of the encoded string. This includes the check character, if any. For Code128, the check character is not present in the returned string.

scanDirection

```
ccAcuBarCodeDefs::ScanDirection scanDirection() const;
```

Returns *eLeftToRight* or *eRightToLeft* to indicate the orientation of the barcode. The orientation must have been computed as a result of calibration.

clientFromImageXform

```
const cc2Xform &clientFromImageXform() const;
```

Returns the client-from-image transform required to translate image coordinates to client coordinates.

time

```
double time() const;
```

Returns the time in seconds required to compute this result.

decodeResult

```
const ccAcuBarCodeResult &decodeResult() const;
```

Returns the result of decoding the barcode using the calibrated barcode properties. The result object must have been computed as a result of calibration.

operator==

```
bool operator== (const ccAcuBarCodeCalibrationResult& that)
const;
```

Returns true if **this* equals *that*, false otherwise. Two **ccAcuBarCodeCalibrationResult** objects are considered equal if all of their corresponding data members are equal except the time. Doubles are compared using **cfRealEq()** (see ch_cvl/math.h) with 1.e-8 as the tolerance.

Parameters

that	**ccAcuBarCodeCalibrationResult** object being compared to the current one.

ccAcuBarCodeCalibrationResult

ccAcquireInfo

```
#include <ch_cvl/acqbase.h>

class ccAcquireInfo;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class stores information about an image acquisition, which can be retrieved when calling **completeAcq()** for that acquisition, or when calling a callback function.

You can examine or test the results of an image acquisition by declaring a **ccAcquireInfo** object to contain the results. You pass this object as a **ccAcqFifo::CompleteArgs** parameter to **completeAcq()** as shown in this code fragment:

```
ccAcquireInfo info;
ccAcqImagePtrh img = fifo->completeAcq
  (ccAcqFifo::CompleteArgs().acquireInfo(&info));
```

To pass a **ccAcquireInfo** object to a callback function, use the **ccAcqFifo** member function **completeInfoCallback()**, **movePartInfoCallback()**, or **overrunInfoCallback()**, as appropriate. These functions register a callback function that the acquisition engine calls at certain times. These callback functions can use member functions of the **ccAcquireInfo** instance passed to them by the acquisition engine.

Constructors/Destructors

ccAcquireInfo

```
ccAcquireInfo();
```

Default constructor, used when creating an object to supply to **completeAcq()**.

Public Member Functions

appTag

```
c_UInt32 appTag() const;
```

Returns the apptag for this acquisition, if one was specified by **ccAcqFifo::start()**.

An apptag is an arbitrary integer optionally applied to each image acquisition initiated with **ccAcqFifo::start()**. An application can assign a tag to each acquisition, and match it to the tag returned by **appTag()** on completion of the acquisition, or on invocation of a callback function.

triggerNum

```
c_UInt32 triggerNum() const;
```

Returns the FIFO's current trigger number count.

A trigger number is a monotonically increasing integer value (excepting integer wraparound) that counts the number of triggers processed by the acquisition FIFO. The trigger number count is maintained automatically by the image acquisition engine, to the best of its ability. You do not need to initialize or start the count of trigger numbers. The trigger number count starts over when the acquisition FIFO object is instantiated.

For some applications, using the trigger number provides an easier method of tracking missed triggers than using **failure().IsMissed()** errors. Tracking the trigger number is particularly useful if the trigger model is semi-automatic or custom, where **failure().IsMissed()** errors require extra calls to **complete()** relative to the number of calls to **start()**.

failure

```
ccAcqFailure failure() const;
```

Returns the **ccAcqFailure** object associated with this image acquisition.

Member functions of **ccAcqFailure** allow you to query the reason an image acquisition failed. See **ccAcqFailure** on page 209. The **ccAcqFailure** object is passed as part of the **ccAcquireInfo** object you associate with each image acquisition.

You can test the success of an acquisition as shown in this code fragment:

```
ccAcquireInfo info;
ccAcqImagePtrh img = fifo->completeAcq
  (ccAcqFifo::CompleteArgs().acquireInfo(&info));
...
if (info.failure())
// Acquisition failed, find out why
    if (info.failure().isIncomplete())
    ...
```

Typedefs

ccCallbackAcqInfo

```
typedef ccCallback1<const ccAcquireInfo&>
    ccCallbackAcqInfo;
```

ccCallbackAcqInfoPtrh

```
typedef ccPtrHandle<ccAcquireInfoCB>
    ccCallbackAcqInfoPtrh;
```

ccAcquireInfo

ccAcuBarCodeDefs

```
#include <ch_cvl/acubar.h>

class ccAcuBarCodeDefs;
```

A name space that holds enumerations used with the Barcode tool classes.

Enumerations

FieldType

```
enum FieldType;
```

Field string specifiers.

Value	Meaning
eLastField	Termination of field map
eAlpha	Character A-Z
eNumeric	Character 0-9
eAlphanumeric	Any alphanumeric
eDash	'-' character
ePeriod	'.' character
eSpace	' ' character
eSemiAlpha	Semichecksum character (0 through 7, and A through H)
eAll	All characters supported by a particular symbology
eOther	All characters specified by *eAll*, but not by any of the other field types

Symbology `enum Symbology;`

Supported symbologies.

Value	Meaning
eCode39	Code 39 or Code 3-of-9; supports all 36 alphanumeric characters, seven additional characters, a variable symbol length, and an optional checksum.
eBC412	Supports alphanumeric characters, A through Z, 0 through 9, dash (-), and the semichecksum characters 0 through 7 and A through H.
eIBM412	Supports alphanumeric characters, A through Z, 0 through 9, dash (-), and the semichecksum characters 0 through 7 and A through H.
eCode128	Supports the entire ASCII character set, plus a packed decimal representation (2 digits per byte).

ScanDirection `enum ScanDirection;`

The scan direction for the symbol. This enumeration can be used to set the intended scan direction and it is used to report the actual scan direction.

Value	Meaning
eLeftToRight	Left-to-right scan.
eRightToLeft	Right-to-left scan.
eBoth	Scan in either direction.

Constants

The following constants specify the maximum number of fields represented by a barcode. This includes the NULL character.

Value	Meaning
kMaxNFieldsCode39 = 21	Maximum number of fields for Code39
kMaxNFieldsIBM412 = 19	Maximum number of fields for IBM412

Value	Meaning
kMaxNFieldsBC412 = 19	Maximum number of fields for BC412
kMaxNFieldsCode128 = 45	Maximum number of fields for Code 128
kMaxNFieldChars = 45	Maximum number of acceptable characters in any field

ccAcuBarCodeDefs

ccAcuBarCodeResult

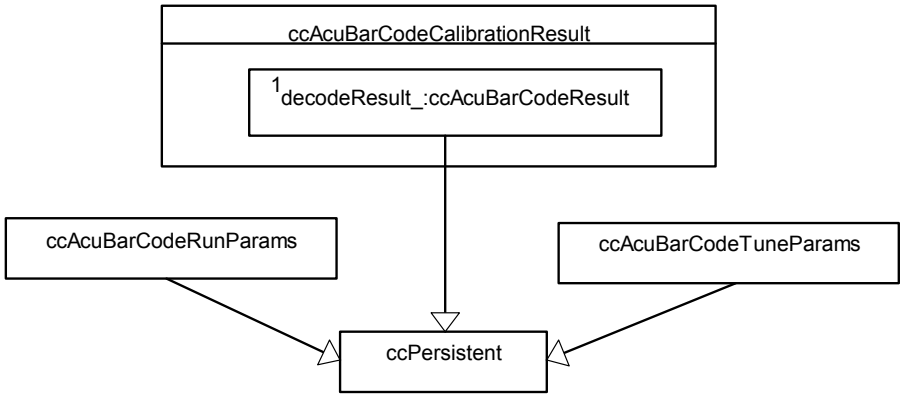
```
#include <ch_cvl/acubar.h>

class ccAcuBarCodeResult : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains results obtained from a single decoding operation executed by a **ccAcuBarCodeTool** object. An instance of this class is contained within a **ccAcuBarCodeCalibrationResult** object.



Constructors/Destructors

ccAcuBarCodeResult

```
ccAcuBarCodeResult();
virtual ~ccAcuBarCodeResult();
```

- `ccAcuBarCodeResult();`

  Constructs a **ccAcuBarCodeResult** object with the data members initialized to the following values:

Parameter	Initial Value
isFound	false
score	0
decoded string	' '
symbol type	*eBC412*
time	0.0
checksumValid	false
scanDirection	*eLeftToRight*
decodedData	Empty vector

- `virtual ~ccAcuBarCodeResult();`

  Destroys an instance of this class.

Operators

operator==

```
bool operator==(const ccAcuBarCodeResult& that) const;
```

Returns true if all of the data members except *time* of this **ccAcuBarCodeResult** object are equal to the data members of a second **ccAcuBarCodeResult** object.

Parameters

that The other parameter set

Public Member Functions

isFound

```
bool isFound() const;
```

Returns true if the string was decoded correctly and the score exceeded the acceptance threshold, false otherwise.

score

```
double score() const;
```

Returns the score that measures the decoding level in the range 0.0 through 1.0. A score of 1.0 indicates perfect decoding, while a score of 0 indicates decoding that was totally unsuccessful.

Notes

For any barcode with a checksum, failure to read the checksum sets the score to 0 (always fails).

decodedString

```
ccCvlString decodedString() const;
```

Returns the decoded string.

decodedData

```
cmStd vector<c_UInt8> decodedData() const;
```

Returns the raw bytes which make up the barcode. For Code 128 symbols, the bytes returned by this function include all of the characters plus the shift and control codes defined in the Code 128 symbology.

symbolType

```
ccAcuBarCodeDefs::Symbology symbolType()
   const;
```

Returns the type of 1D Symbology that was decoded to produce this result.

time

```
double time() const;
```

Returns the time required in seconds to compute this result.

checksumValid

```
bool checksumValid() const;
```

Returns true if *checksum* is valid. As defined under “Some Useful Definitions” in the Barcode Tool chapter of the *CVL Vision Tools Guide*, a checksum is a “character used in calculating a check for correct decoding.”

Notes

For any barcode with a checksum, failure to read the checksum sets the score to 0, which causes the *decode()* operation to fail. **checksumValid()** returns true only if a checksum was present, and the tool was successful in reading the checksum irrespective of whether the tool passed the acceptance threshold.

scanDirection

```
ccAcuBarCodeDefs::ScanDirection scanDirection() const;
```

Returns the direction in which the symbol was scanned. This function returns one of the following values:

ccAcuBarCodeDefs::eRightToLeft
ccAcuBarCodeDefs::eLeftToRight

ccAcuBarCodeRunParams

```
#include <ch_cvl/acusymbl.h>

class ccAcuBarCodeRunParams: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccAcuBarCodeRunParams** class contains a set of common parameters required to support the Barcode tool. After you have set up a **ccAcuBarCodeRunParams** object, you pass it to **ccAcuBarCodeTool::decode()**, which performs the decoding operation.

You can set up the lighting parameters yourself, or you can use **ccAcuBarCodeTool::tune()** to iterate over several possibilities to create a set of lighting parameters for you.

Constructors/Destructors

ccAcuBarCodeRunParams

```
ccAcuBarCodeRunParams(
  ccAcuBarCodeDefs::Symbology symbolType =
    ccAcuBarCodeDefs::eBC412,
  ccAcuBarCodeDefs::FieldType defaultType =
    ccAcuBarCodeDefs::eAlphanumeric,
  double accept = 0.5,
  ccAcuBarCodeDefs::ScanDirection scanDirection =
    ccAcuBarCodeDefs::eLeftToRight);
```

Constructs a **ccAcuBarCodeRunParams** object, and initializes the symbology type to that specified by *symbolType* and the field strings at all positions to those specified by *defaultType*. Requires that *defaultType* be a valid field type for the specified symbology.

The other parameters are initialized to the following values:

- checksum is false for Code39 and true for Code128, BC412, and IBM412, which have built-in error-correcting checksums.
- *lightPower* = 0.0.
- *brightFieldPowerRatio* = 0.0.

Parameters

symbolType The format for encoding and decoding a specific type of barcode.

defaultType The type of character that each field may contain.

accept The acceptance threshold. Valid range: 0.0 through 1.0. The acceptance threshold is the percentage of scan lines crossing the barcode that must correctly decode for the read to pass.

scanDirection The direction in which to scan the barcode. If you specify *ccAcuBarCodeDefs::eBoth*, then the tool attempts to decode the barcode in both left-to-right and right-to-left directions; it returns the best score. *scanDirection* must be one of the following values:

ccAcuBarCodeDefs::eRightToLeft
ccAcuBarCodeDefs::eLeftToRight
ccAcuBarCodeDefs::eBoth

Notes

BC412 and IBM412 symbologies support the *ccAcuBarCodeDefs::eLastField*, *ccAcuBarCodeDefs::eAlpha*, *ccAcuBarCodeDefs::eNumeric*, *ccAcuBarCodeDefs::eAlphanumeric*, *ccAcuBarCodeDefs::eDash, ccAcuBarCodeDefs::eSemiAlpha*, and *ccAcuBarCodeDefs::eAll* field types. Code39 and Code128 support all of the aforementioned plus *ccAcuBarCodeDefs::ePeriod, ccAcuBarCodeDefs::eSpace*, and *ccAcuBarCodeDefs::eOther*.

Throws

*ccAcuBarCodeDefs::BadParam*s
An invalid field type is specified for the symbology type, or *accept* is out of range 0.0 through 1.0.

Public Member Functions

accept

```
double accept() const;

void accept(double acceptVal);
```

- `double accept() const;`

Gets the accept threshold, which is the percentage of scan lines crossing the barcode that must correctly decode for the read to pass.

- `void accept(double acceptVal);`

  Sets the accept threshold.

  Parameters

  acceptVal The accept threshold; requires that *acceptVal* be between 0.0 and 1.0 inclusive.

  Throws

  ccAcuBarCodeDefs::BadParams
  The value being set is outside the valid range.

changeSymbolAndFieldType

```
void changeSymbolAndFieldType (
  ccAcuBarCodeDefs::Symbology symbolType,
  ccAcuBarCodeDefs::FieldType type);
```

Changes the type of 1D Symbology to be decoded and sets the field strings at all positions to the type specified by *type*.

Parameters

symbolType The format for encoding and decoding a specific type of barcode.

type The type of character that each field may contain.

Throws

ccAcuBarCodeDefs::BadParams
An invalid field type is specified for the symbology type.

Notes

BC412 and IBM412 symbologies support the *ccAcuBarCodeDefs::eLastField*, *ccAcuBarCodeDefs::eAlpha*, *ccAcuBarCodeDefs::eNumeric*, *ccAcuBarCodeDefs::eAlphanumeric*, *ccAcuBarCodeDefs::eDash, ccAcuBarCodeDefs::eSemiAlpha*, and *ccAcuBarCodeDefs::eAll* field types. Code39 and Code128 support all of the aforementioned plus *ccAcuBarCodeDefs::ePeriod, ccAcuBarCodeDefs::eSpace*, and *ccAcuBarCodeDefs::eOther*.

ccAcuBarCodeRunParams::changeSymbolAndFieldType() invalidates all previously set field string specifications for the object.

Whenever the symbology of a **ccAcuBarCodeRunParams** object is changed to *ccAcuBarCodeDefs::eBC412, ccAcuBarCodeDefs::eIBM412*, *ccAcuBarCodeDefs::eCode39*, or *ccAcuBarCodeDefs::eCode128*, the checksum is enabled.

checksum

```
bool checksum() const;
void checksum(bool val);
```

- `bool checksum() const;`

  Gets the checksum enable state.

- `void checksum(bool val);`

  Sets the checksum enable state. (For BC412, IBM412, and Code128 symbologies, *val* is always true.)

  Parameters

  val — True or false depending on whether the tool should perform a checksum operation.

  Notes

  For any symbology that has an optional checksum, you can set this parameter to true or false. For other symbologies, such as BC412 and IBM412 which have built-in error-correcting checksums, it is always set to true. You cannot toggle this value.

  Throws

  ccAcuBarCodeDefs::BadParams — The given symbology is *ccAcuBarCodeDefs::eBC412*, *ccAcuBarCodeDefs::eIBM412*, or *ccAcuBarCodeDefs::eCode128*, and the setter is passed false.

brightFieldPowerRatio

```
double brightFieldPowerRatio() const;
void brightFieldPowerRatio(double val);
```

- `double brightFieldPowerRatio() const;`

  Returns the bright field power ratio for this object. The bright field power ratio is the portion (in the range 0.0 through 1.0) of the total light energy allocated to the bright field lights.

- `void brightFieldPowerRatio(double val);`

  Sets the bright field power ratio for this object. The bright field power ratio is the portion (in the range 0.0 through 1.0) of the total light energy allocated to the bright field lights.

Parameters

val The power ratio. Must be in the range 0.0 (full darkfield) through 1.0 (full brightfield).

Throws

ccAcuBarCodeDefs::BadParams
val was out of range.

fieldString

```
cmStd string fieldString (c_UInt32 position) const;

void fieldString(c_UInt32 position,
 const cmStd string &str);

void fieldString(c_UInt32 position,
 ccAcuBarCodeDefs::FieldType type);
```

- `cmStd string fieldString (c_UInt32 position) const;`

Returns the acceptable characters in the given field position.

Parameters

position The character position. Character positions must be between the following values:

0 and *kMaxNFieldsCode39* - 1 inclusive for symbology *ccAcuBarCodeDefs::eCode39*
0 and *kMaxNFieldsBC412* - 1 inclusive for symbology *ccAcuBarCodeDefs::eBC412*
0 and *kMaxNFieldsIBM412* - 1 inclusive for symbology *ccAcuBarCodeDefs::eIBM412*
0 and *kMaxNFieldsCode128* - 1 inclusive for symbology *ccAcuBarCodeDefs::eCode128*

Throws

ccAcuBarCodeDefs::BadParams
The position specified is invalid for the symbology type of this **ccAcuBarCodeRunParams** object.

- `void fieldString(c_UInt32 position,`
  `const cmStd string &str);`

Sets the field string at certain positions.

Parameters

position The character position. Character positions must be between the following values:

0 and *kMaxNFieldsCode39* - 1 inclusive for symbology *ccAcuBarCodeDefs::eCode39*
0 and *kMaxNFieldsBC412* - 1 inclusive for symbology *ccAcuBarCodeDefs::eBC412*
0 and *kMaxNFieldsIBM412* - 1 inclusive for symbology *ccAcuBarCodeDefs::eIBM412*
0 and *kMaxNFieldsCode128* - 1 inclusive for symbology *ccAcuBarCodeDefs::eCode128*

str — Valid characters for the symbology type.

Throws

*ccAcuBarCodeDefs::BadParam*s
Position is not in the valid range, or the string contains invalid characters.

- `void fieldString(c_UInt32 position, ccAcuBarCodeDefs::FieldType type);`

Sets the field string at a particular position to a predefined *fieldType* value.

Parameters

position — The character position. Character positions must be between the following values:

0 and *kMaxNFieldsCode39* - 1 inclusive for symbology *ccAcuBarCodeDefs::eCode39*
0 and *kMaxNFieldsBC412* - 1 inclusive for symbology *ccAcuBarCodeDefs::eBC412*
0 and *kMaxNFieldsIBM412* - 1 inclusive for symbology *ccAcuBarCodeDefs::eIBM412*
0 and *kMaxNFieldsCode128* - 1 inclusive for symbology *ccAcuBarCodeDefs::eCode128*

type — Valid field type for the symbology type of this **ccAcuBarCodeRunParams** object.

Notes

BC412 and IBM412 symbologies support the *ccAcuBarCodeDefs::eLastField*, *ccAcuBarCodeDefs::eAlpha*, *ccAcuBarCodeDefs::eNumeric*, *ccAcuBarCodeDefs::eAlphanumeric*, *ccAcuBarCodeDefs::eDash, ccAcuBarCodeDefs::eSemiAlpha*, and *ccAcuBarCodeDefs::eAll* field types. Code39 and Code128 support all of the aforementioned plus *ccAcuBarCodeDefs::ePeriod, ccAcuBarCodeDefs::eSpace*, and *ccAcuBarCodeDefs::eOther*.

Throws

ccAcuBarCodeDefs::BadParams

Position is not in the valid range, or the field type does not agree with the symbology type of this **ccAcuBarCodeRunParams** object.

lightPower

```
double lightPower() const;

void lightPower(double val);
```

- `double lightPower() const;`

  Gets the light power to be used.

- `void lightPower(double val);`

  Sets the light power to be used.

  Parameters

  val The light power. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuBarCodeDefs::BadParams

  val lies outside the valid range.

symbolType

```
ccAcuBarCodeDefs::Symbology symbolType() const;
```

Gets the type of 1D Symbology to be decoded using the parameters contained in the **ccAcuBarCodeRunParams** object.

scanDirection

```
ccAcuBarCodeDefs::ScanDirection scanDirection() const;

void scanDirection(ccAcuBarCodeDefs::ScanDirection
   scanDirection);
```

- `ccAcuBarCodeDefs::ScanDirection scanDirection() const;`

  Returns the scan direction of this **ccAcuBarCodeRunParams** object. The function returns one of the following values:

  ccAcuBarCodeDefs::eRightToLeft
  ccAcuBarCodeDefs::eLeftToRight
  ccAcuBarCodeDefs::eBoth

- `void scanDirection(ccAcuBarCodeDefs::ScanDirection scanDirection);`

  Sets the scan direction of this **ccAcuBarCodeRunParams** object.

  Parameters

  scanDirection The direction in which to scan the barcode. If you specify *ccAcuBarCodeDefs::eBoth*, then the tool attempts to decode the barcode in both left-to-right and right-to-left directions; it returns the best score. *scanDirection* must be one of the following values:

  ccAcuBarCodeDefs::eRightToLeft
  ccAcuBarCodeDefs::eLeftToRight
  ccAcuBarCodeDefs::eBoth

Operators

operator== `bool operator==(const ccAcuBarCodeRunParams& that) const;`

Returns true if these **ccAcuBarCodeRunParams** are the same as another set of **ccAcuBarCodeRunParams**. Two parameter sets are considered equal if all of their attributes are the same.

Parameters

that The other parameter set

ccAcuBarCodeTool

```
#include <ch_cvl/acubar.h>

class ccAcuBarCodeTool;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class represents the base class for the 1D symbology tools. Tools for decoding specific 1D symbologies are derived from the base class.

The class **ccAcuBarCodeTool** and the related classes **ccAcuBarCodeRunParams**, **ccAcuBarCodeTuneParams**, and **ccAcuBarCodeResult** support the following two types of 1D barcode decoders:

- Decoders used for BC412 and IBM412 symbologies, which may require an intermediate tuning step based on light settings to estimate optimal parameters.
- Decoders used for the Code39 and Code 128 symbologies, which do not normally require any tuning prior to decoding except in some special cases (for example, when reading wafers).

To create the Barcode tool, you create a **ccAcuBarCodeTool** object and then perform the following steps:

1. Create a **ccAcuBarCodeResult** object.

   ```
   ccAcuBarCodeResult result;
   ```

2. Create a **ccAcuBarCodeRunParams** object.

   ```
   ccAcuBarCodeRunParams bestParams(ccAcuBarCodeDefs::eIBM412);
   bestParams.lightPower(0.5);
   bestParams.fieldString(7, ccAcuBarCodeDefs::eLastField);
   ```

3. Create a **ccAcuBarCodeTuneParams** object.

   ```
   ccAcuBarCodeTuneParams stTuneParams;
   ```

4. Extract the region of interest to be used for tuning.

```
cc8120& fg = cc8120::get(0);
const ccStdVideoFormat& fmt =
        ccStdVideoFormat::getFormat(cmT("Sony XC75 640x480"))
ccStdGreyAcqFifo *fifo = fmt.newAcqFifo(fg);

fifo->properties().triggerEnable( true);
cc2Xform xform(cc2Matrix(1, 0, 0, 1), cc2Vect(0, 0));
ccAffineRectangle affrect(cc2Vect(110, 180),
          cc2Vect(210, 190), cc2Vect(110, 290));
ccAffineSamplingParams affparams(affrect, 100, 100);
```

5. Call **tune()** to perform the tuning step.

```
ccAcuBarCodeTool mbct;
mbct.tune (*fifo, xform, affparams, stTuneParams,
   bestParams, result);
```

To perform a barcode decoding operation, perform the following steps:

1. Acquire the image.

```
ccPelBuffer_const<c_UInt8> img = fifo->complete();
```

Note Make sure that the acquired image includes the entire barcode, including the quiet zone. For more information on quiet zones, see the chapter *The Barcode Tool* in the *CVL Vision Tools Manual*.

2. Create a **ccAcuBarCodeResult** object.

```
ccAcuBarCodeResult result;
```

3. Create a **ccAcuBarCodeRunParams** object.

```
ccAcuBarCodeRunParams params;
```

4. If necessary, change symbol and field type.

```
params.changeSymbolAndFieldType(ccAcuBarCodeDefs::eIBM412,
          ccAcuBarCodeDefs::eAlpha);
params.fieldString(0, ccAcuBarCodeDefs::eAlphanumeric);
params.fieldString(7, ccAcuBarCodeDefs::eLastField);
params.accept(0.1);
```

5. Decode the barcode.

```
mbct.decode (img, params, result);
```

For more detailed information about decoding barcodes, see the chapter *The Barcode Tool* in the *CVL Vision Tools Manual*.

In addition to the tune and decode functions, the Barcode tool supports a calibrate function, which automatically determines the properties and location of a barcode in an image. After a successful calibration, the **ccAcuBarCodeCalibrationResult** class holds the result.

To calibrate the Barcode tool, perform the following steps:

1. Acquire the image.

```
ccPelBuffer_const<c_UInt8> img = fifo->complete();
```

Note Make sure that the acquired image includes the entire barcode, including the quiet zone. For more information on quiet zones, see the chapter *The Barcode Tool* in the *CVL Vision Tools Manual*

2. Create a **ccAcuBarCodeRunParams** object to specify the runtime parameters for each symbology to be included in the calibration process. For example, to calibrate the Barcode tool for Code39 and BC412 symbologies, create two **ccAcuBarCodeRunParams** objects, one for eCode39 and one for eBC412. By creating only these two objects, you can exclude the other supported symbologies (for example, IBM412 and Code128) from the calibration process.

```
ccAcuBarCodeRunParams code39Params;
ccAcuBarCodeRunParams bc412Params;
```

3. Set the runtime parameters for each symbology, if desired. Runtime parameters can include *accept threshold*, *character set* for each field, *length* of encoded string, *scan direction*, and presence of *checksum* character.

   Set the *length* of the encoded string using the *eLastField* character to mark the end of the string. Not specifying an *eLastField* character in the field map implies that the encoded string length is to be determined.

   Setting the *scan direction* to *eBoth* implies that the barcode scan direction is to be determined.

   See the chapter on the **ccAcuBarCodeRunParams** class for details.

4. Create a vector of the **ccAcuBarCodeRunParams** objects created in the last step.

```
cmStd vector<ccAcuBarCodeRunParams> runParamsSet[2];
runParamsSet[0] = code39Params;
runParamsSet[1] = bc412Params;
```

5. Create a **ccAcuBarCodeCalibrationResult** object.

```
ccAcuBarCodeCalibrationResult calResult;
```

6. Calibrate the Barcode tool.

```
ccAcuBarCodeTool mbct;
mbct.calibrate(img, runParamsSet, calResult);
```

Constructors/Destructors

ccAcuBarCodeTool

```
ccAcuBarCodeTool();

virtual ~ccAcuBarCodeTool();
```

- `ccAcuBarCodeTool();`

  Default constructor.

- `virtual ~ccAcuBarCodeTool();`

  Destroys instances of this class.

Public Member Functions

decode

```
virtual void decode (ccGreyAcqFifo &fifo,
  const cc2Xform &xform,
  const ccAffineSamplingParams &affParams,
  const ccAcuBarCodeRunParams &runParams,
  ccAcuBarCodeResult &result, bool autoRetry = true) const;

virtual void decode (
  const ccPelBuffer_const<c_UInt8> &image,
  const ccAcuBarCodeRunParams &runParams,
  ccAcuBarCodeResult &result, bool autoRetry = true) const;
```

- ```
  virtual void decode (ccGreyAcqFifo &fifo,
    const cc2Xform &xform,
    const ccAffineSamplingParams &affParams,
    const ccAcuBarCodeRunParams &runParams,
    ccAcuBarCodeResult &result, bool autoRetry = true) const;
  ```

  Decodes the barcode located within the region of interest (ROI) specified by *affParams* and returns the results of the decode operation in the object specified by *result*.

  The *fifo* object acquires an image containing the barcode symbol. The light properties of the FIFO are based on the light power and dark level parameters contained in the *runParams* object. You set these parameters prior to acquiring the image. The *xform* object specifies the *clientFromImage* transform of the acquired image. The affine sampling parameters are used to extract the region of interest containing the target
  ```

barcode symbol. The tool then uses this region of interest for decoding the string. When the decoding operation finishes, the properties of the FIFO are left in the state specified by *runParams*.

If the decode operation returns a score for the barcode below the accept threshold you specify in *runParams*, the tool will automatically attempt to decode the barcode using an advanced decoding strategy. If the tool invokes the advanced decoding strategy, the decode will take about 10 times longer to perform, but the advanced strategy may be able to handle barcodes that could not otherwise be decoded.

If you need to ensure that calls to decode take a consistent amount of time, you can prevent the tool from automatically invoking the advanced decode strategy by specifying a value of false for the *autoRetry* argument to this function.

Requires that *xform* be nonsingular and the region of interest be nonnull.

The decode operation supports timeouts via the use of ccTimeout.

Parameters

fifo	An acquisition FIFO.
xform	A coordinate transformation to convert pixel coordinates to client coordinates.
affParams	Affine sampling parameters used to extract the region of interest containing the target barcode symbol.
runParams	A set of parameters used to support the decoding operation including symbol type, field type, and accept threshold.
result	Results obtained from a single decoding operation including score, the decoded string, and the result of the checksum operation.
autoRetry	Specify false to prevent the tool from automatically retrying failed decodes using the advanced decoding strategy. Specify true to allow the tool to automatically retry failed decodes.

Notes

To extract the ROI, the tool performs a windowing operation based on the affine rectangle if you specify the following affine sampling parameters:

- No subsampling
- Image alignment of image coordinates

Otherwise, the tool samples the acquired image as specified.

Throws

ccAcuBarCodeDefs::BadParams

The *xform* is singular or the region of interest specified by *affParams* is null or the field string in *runParams* does not contain a *ccAcuBarCodeDefs::eLastField* character marking the end of the string.

ccAcuBarCodeDefs::NotImplemented

The *symbolType* of *runParams* is not *eBC412*, *eIBM412*, *eCode39*, or *eCode128*.

- ```
  virtual void decode (
    const ccPelBuffer_const<c_UInt8> &image,
    const ccAcuBarCodeRunParams &runParams,
    ccAcuBarCodeResult &result, bool autoRetry = true) const;
  ```

Decodes the string from the source image, and returns the result in the specified result object. Requires that the image be bound, and its window be adjusted to fit the region of interest containing the target barcode symbol and the quiet zone.

If the decode operation returns a score for the barcode below the accept threshold you specify in *runParams*, the tool will automatically attempt to decode the barcode using an advanced decoding strategy. If the tool invokes the advanced decoding strategy, the decode will take about 10 times longer to perform, but the advanced strategy may be able to handle barcodes that could not otherwise be decoded.

If you need to ensure that calls to **decode()** take a consistent amount of time, you can prevent the tool from automatically invoking the advanced decode strategy by specifying a value of false for the *autoRetry* argument to this function.

The decode operation supports timeouts via the use of ccTimeout.

Parameters

image — The pel buffer that contains the image being read.

runParams — A set of parameters used to support the decoding operation including symbol type, field type, and accept threshold.

result — Results obtained from a single decoding operation including score, the decoded string, and the result of the checksum operation.

autoRetry — Specify false to prevent the tool from automatically retrying failed decodes using the advanced decoding strategy. Specify true to allow the tool to automatically retry failed decodes.
  ```

Throws

ccAcuBarCodeDefs::BadParams
The image is not bound or the field string in *runParams* does not contain an *ccAcuBarCodeDefs::eLastField* character marking the end of the string.

ccAcuBarCodeDefs::NotImplemented
The *symbolType* of *runParams* is not *eBC412*, *eIBM412*, *eCode39*, or *eCode128*.

---

calibrate

```
virtual void calibrate (
  const ccPelBuffer_const<c_UInt8> &image,
  ccAcuBarCodeCalibrationResult &result,
  const cmStd vector<ccAcuBarCodeDefs::Symbology>
    &symbolTypes =
    cmStd vector<ccAcuBarCodeDefs::Symbology>());

virtual void calibrate (
  const ccPelBuffer_const<c_UInt8> &image,
  const cmStd vector<ccAcuBarCodeRunParams> &runParamsSet,
  ccAcuBarCodeCalibrationResult &result);
```

-
```
virtual void calibrate (
  const ccPelBuffer_const<c_UInt8> &image,
  ccAcuBarCodeCalibrationResult &result,
  const cmStd vector<ccAcuBarCodeDefs::Symbology>
&symbolTypes =
  cmStd vector<ccAcuBarCodeDefs::Symbology>());
```

Given an image containing a barcode, this calibrate method determines all of the barcode properties using an accept threshold of 0.5 and disabling the checksum for eCode39, for which the checksum is optional. Calibration determines the barcode symbology (Code39, BC412, IBM412, or Code128), encoded string length, orientation (left-to-right or right-to-left), pitch, and location (ROI in the image that encloses the barcode).

Parameters

image — Pelbuffer containing the barcode to be used for calibration.

result — Result object containing information on the result of the calibration.

symbolTypes — Specifies a vector of the symbologies to be included in the calibration. For example, to exclude eCode128 from calibration, specify a vector of the other three supported symbol types (eCode39, eBC412, and eIBM412). Specifying an empty vector for symbolTypes causes the calibrate method to take all supported symbologies into consideration.

Throws

ccAcuBarCodeDefs::NotImplemented
The symbology specified by the vector of symbol types is not one of eCode39, eBC412, eIBM412, or eCode128.

ccAcuBarCodeDefs::BadParams
Two or more elements in the symbolTypes vector correspond to the same symbology.

ccAcuBarCodeDefs::BadImage
The image is not bound or does not satisfy the minimum window size requirement of 32x10 pixels.

- ```
  virtual void calibrate (
    const ccPelBuffer_const<c_UInt8> &image,
    const cmStd vector<ccAcuBarCodeRunParams> &runParamsSet,
    ccAcuBarCodeCalibrationResult &result);
  ```

Given an image containing a barcode, the calibrate method determines the barcode symbology (Code39, BC412, IBM412, or Code128), encoded string length, orientation (left-to-right or right-to-left), pitch, and location (that is, the ROI in the image that encloses the barcode).

Parameters

image — Pelbuffer containing the barcode to be used for calibration.

runParamsSet — Specifies a vector of runtime parameters for each symbology to be included in the calibration process. For example, to exclude eIBM412 and eCode128 from calibration, specify a vector of two runparams objects, one for eCode39 and one for eBC412, each containing a set of parameters to control the calibration of the corresponding symbology. For each symbology, you can control any or all of the following runtime parameters:
- *accept threshold*
- *character set* for each field
- *length* of the encoded string (by using the eLastField character to mark the end of the string; not specifying an eLastField character in the field map implies that the encoded string length is to be determined)
- *scan direction* (eBoth implies that the barcode orientation is to be determined)
- presence or absence of *checksum* character

result — Result object containing information on the result of the calibration.
  ```

Throws

ccAcuBarCodeDefs::NotImplemented
The symbology specified by the *runParamsSet* vector is not one of eCode39, eBC412, eIBM412, or eCode128.

ccAcuBarCodeDefs::BadParams
The *runParamsSet* vector is empty, or two or more elements in the vector correspond to the same symbology.

ccAcuBarCodeDefs::BadImage
The image is not bound or does not satisfy the minimum window size requirement of 32x10 pixels.

---

tune

```
virtual void tune (
  ccGreyAcqFifo &fifo,
  const cc2Xform &xform,
  const ccAffineSamplingParams &affParams,
  const ccAcuBarCodeTuneParams &startTuneParams,
  ccAcuBarCodeRunParams &bestParams,
  ccAcuBarCodeResult &result);

virtual void tune(
  const ccAcuBarCodeTuneParams &startTuneParams,
  ccAcuBarCodeRunParams &bestParams,
  ccAcuBarCodeResult &result);
```

-
```
virtual void tune (
  ccGreyAcqFifo &fifo,
  const cc2Xform &xform,
  const ccAffineSamplingParams &affParams,
  const ccAcuBarCodeTuneParams &startTuneParams,
  ccAcuBarCodeRunParams &bestParams,
  ccAcuBarCodeResult &result);
```

Estimates optimal decoding parameters for the Barcode tool based on various light settings, which you specify in *startTuneParams*.

Images are acquired using the FIFO object specified by *fifo*. The *xform* object specifies the *clientFromImage* transform for the acquired image. The affine sampling parameters are used to extract the region of interest to be used for tuning. The optimal light parameters are returned via the *bestParams* object. The result of decoding using optimal parameters is returned via the *result* object. Upon completion of the tune operation, the tool sets the light properties of the FIFO object to the optimal values returned via *bestParams*.

Requires that *xform* be nonsingular and the region of interest be nonnull.

The tune operation supports timeouts via the use of ccTimeout.

Parameters

fifo	An acquisition FIFO.
xform	A coordinate transformation to convert pixel coordinates to client coordinates.
affParams	Affine sampling parameters used to extract the region of interest containing the target barcode symbol.
startTuneParams	Tuning parameters including dark field and bright field properties.
bestParams	A set of parameters used to support the decoding operation including symbol type, field type, and accept threshold.
result	Results obtained from a single decoding operation including the score, the decoded string, and the result of the checksum operation.

Notes

To extract the ROI, the tool performs a windowing operation based on the affine rectangle if you specify the following affine sampling parameters:

- No subsampling
- Image alignment of image coordinates

Otherwise, the tool samples the acquired image as specified.

Use this function only if you need to tune light parameters in applications such as Wafer ID where the symbology type is either BC412 or IBM412.

Throws

ccAcuBarCodeDefs::BadParams
: The *xform* is singular, the region of interest specified by *affparams* is null, or the field string in *bestParams* does not contain an *ccAcuBarCodeDefs::eLastField* character marking the end of the string.

ccAcuBarCodeDefs::NotImplemented
: The *symbolType* of *bestParams* is not *eBC412*, *eIBM412*, *eCode39*, or *eCode128*.

- ```
  virtual void tune(
    const ccAcuBarCodeTuneParams &startTuneParams,
    ccAcuBarCodeRunParams &bestParams,
    ccAcuBarCodeResult &result);
  ```

  Estimates optimal decoding parameters for the Barcode tool based on various light settings, which you specify in *startTuneParams*.

  You supply the images used for tuning by overriding the **ccAcuBarCodeTool::acquireImage()** function.

  The optimal light parameters are returned via the *bestParams* object. The result of decoding using optimal parameters is returned via the *result* object. Upon completion of the tuning operation, the tool sets the light properties of the FIFO object to the optimal values returned via *bestParams*.

  Requires that *xform* be nonsingular and the region of interest be nonnull.

  The tune operation supports timeouts via the use of ccTimeout.

  Parameters

  startTuneParams Tuning parameters including dark field and bright field properties.

  bestParams A set of parameters used to support the decoding operation including symbol type, field type, and accept threshold.

  result Results obtained from a single decoding operation including the score, the decoded string, and the result of the checksum operation.

  Notes

  Use this function only if you need to tune light parameters in applications such as Wafer ID where the symbology type is either BC412 or IBM412.

  Throws

  ccAcuBarCodeDefs::NotImplemented
  The *symbolType* of *bestParams* is not *eBC412*, *eIBM412*, *eCode39*, or *eCode128*.

  ccAcuBarCodeDefs::BadParams
  The field string in *bestParams* does not contain an *ccAcuBarCodeDefs::eLastField* character marking the end of the string.
  ```

acquireImage

```
virtual ccPelBuffer<c_UInt8> acquireImage(
  const ccAcuBarCodeRunParams& currentParams);
```

You should not call this function directly. To use this function, override it in a class that you derive from **ccAcuBarCodeTool**.

This function lets you tune lighting parameters without using an acquisition FIFO to acquire images. When you use the second overload of **ccAcuBarCodeTool::tune()**, this function is called to obtain images. Your overload of this function should return the image to use for tuning.

Parameters

currentParams The current acquisition parameters.

---

update

```
virtual bool update () const;

virtual bool update(
  const ccAcuBarCodeRunParams &currentParams,
  const ccAcuBarCodeRunParams &bestParams,
  const ccAcuBarCodeResult &currentResult,
  const ccAcuBarCodeResult &bestResult)
```

- ```
  virtual bool update () const;
  ```

  Returns bool, indicating whether the tuning process should continue or not.

- ```
  virtual bool update(
    const ccAcuBarCodeRunParams &currentParams,
    const ccAcuBarCodeRunParams &bestParams,
    const ccAcuBarCodeResult &currentResult,
    const ccAcuBarCodeResult &bestResult)
  ```

  You should not call this function directly. To use this function, override it in a class that you derive from **ccAcuBarCodeTool**.

  Provides a hook that lets you update intermediate **tune()** results. The tuning process continues as long as this function returns true. The default **update()** function always returns true.

  Notes

  A user-overloaded **update()** function may abort tuning by returning false.

  Parameters

  currentParams The current acquisition parameters.

  bestParams The best acquisition parameters.

  currentResult The results of decoding using *currentParams*.

bestResult	The results of decoding using *bestParams*.

ccAcuBarCodeTool

ccAcuBarCodeTuneParams

```
#include <ch_cvl/acubar.h>

class ccAcuBarCodeTuneParams : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class provides tuning parameters for the tuning process of the Barcode tool. You pass the **ccAcuBarCodeTuneParams** object to **ccAcuBarCodeTool::tune()**. That function uses the information in the tuning parameters to generate a **ccAcuBarCodeRunParams** object that contains the optimal light parameters for the decoding operation.

Constructors/Destructors

ccAcuBarCodeTuneParams

```
ccAcuBarCodeTuneParams (
  bool enableDark = true,
  double darkLow = 0.062745,
  double darkHigh = 1.0,
  double darkStep = 0.12157,
  bool enableBright = true,
  double brightLow = 0.062745,
  double brightHigh = 1.0,
  double brightStep = 0.12157);
```

Creates a tune-parameters object with the specified values.

Parameters

enableDark	True to enable darkfield tuning.
darkLow	The minimum light power for darkfield tuning. Must be in the range 0.0 through 1.0.
darkHigh	The maximum light power for darkfield tuning. Must be in the range 0.0 through 1.0.
darkStep	The darkfield step value. Must be in the range 0.0 through 1.0.
enableBright	True to enable brightfield tuning.

brightLow The minimum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

brightHigh The maximum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

brightStep The brightfield step value. Must be in the range 0.0 through 1.0.

Throws

ccAcuBarCodeDefs::BadParams
darkLow, darkHigh, darkStep, brightLow, brightHigh, brightStep are set to be outside the valid range 0.0 through 1.0.

Notes

The default dark field and light field values are set as specified.

Operators

operator==

```
bool operator==(const ccAcuBarCodeTuneParams& that) const;
```

Returns true if these **ccAcuBarCodeTuneParams** are the same as another set of **ccAcuBarCodeTuneParams**. Two parameter sets are considered equal if all of their attributes are the same.

Parameters

that The other parameter set

Public Member Functions

brightLow

```
double brightLow() const;
void brightLow(double val);
```

- `double brightLow() const;`

Returns the minimum light power for brightfield tuning.

- `void brightLow(double val);`

Sets the minimum light power for brightfield tuning.

Parameters

val The minimum light power for brightfield tuning. Must be in the range 0.0 through 1.0 where the light power increases monotonically.

Throws
ccAcuBarCodeDefs::BadParams
val is out of range.

brightHigh

```
double brightHigh() const;
void brightHigh(double val);
```

- `double brightHigh() const;`

Returns the maximum light power for brightfield tuning.

- `void brightHigh(double val);`

Sets the maximum light power for brightfield tuning.

Parameters

val	The maximum light power for brightfield tuning. Must be in the range 0.0 through 1.0 where the light power increases monotonically.

Throws
ccAcuBarCodeDefs::BadParams
val is out of range.

brightStep

```
double brightStep() const;
void brightStep(double val);
```

- `double brightStep() const;`

Returns the brightfield step for tuning.

- `void brightStep(double val);`

Sets the brightfield step for tuning.

Parameters

val	The brightfield step value. Must be in the range 0.0 through 1.0 where the light power increases monotonically.

Throws
ccAcuBarCodeDefs::BadParams
val is out of range.

ccAcuBarCodeTuneParams

darkLow

```
double darkLow() const;
void darkLow(double val);
```

- `double darkLow() const;`

  Returns the minimum light power for darkfield tuning.

- `void darkLow(double val);`

  Sets the minimum light power for darkfield tuning.

  Parameters

  val — The minimum light power for darkfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuBarCodeDefs::BadParams
  val is out of range.

darkHigh

```
double darkHigh() const;
void darkHigh(double val);
```

- `double darkHigh() const;`

  Returns the maximum light power for darkfield tuning.

- `void darkHigh(double val);`

  Sets the maximum light power for darkfield tuning.

  Parameters

  val — The maximum light power for darkfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuBarCodeDefs::BadParams
  val is out of range.

darkStep

```
double darkStep() const;
void darkStep(double val);
```

- `double darkStep() const;`

  Returns the darkfield step for tuning.

- `void darkStep(double val);`

  Sets the darkfield step for tuning.

  Parameters

  val The darkfield step value. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuBarCodeDefs::BadParams
  val is out of range.

enableBright

```
bool enableBright() const;
void enableBright(bool val);
```

- `bool enableBright() const;`

  Returns true if brightfield tuning is enabled.

- `void enableBright(bool val);`

  Enables or disables brightfield tuning.

  Parameters

  val True if brightfield tuning is enabled. False otherwise.

enableDark

```
bool enableDark() const;
void enableDark(bool val);
```

- `bool enableDark() const;`

  Returns true if darkfield tuning is enabled.

- `void enableDark(bool val);`

  Enables or disables darkfield tuning.

Parameters

val	True if darkfield tuning is enabled. False otherwise.

ccAcuRead

```
#include <ch_cvl/acuread.h>

class ccAcuRead;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class provides access to basic OCR reading and tuning functions. In your application, you would create an instance of this class and use the **read()** function to read a string. You can use the **tune()** function to obtain the optimum reading parameters for your application. The mechanism for enabling Non-Linear OCR is to pass a **ccAcuReadFont** that has been initialized with a nonlinear font to the **read()** or **tune()** function.

You can control some aspects of the tuning function by creating a class derived from this class and overriding the functions **update()** and **userPreprocessAcquiredImage()**.

For more detailed information about OCR, reading, and tuning, see the chapter *acuRead* in the *CVL Vision Tools Guide*.

Constructors/Destructors

You can use the default constructor and destructor to create and destroy instances of this class.

Public Member Functions

tune

```
void tune(const ccPelBuffer_const<c_UInt8> &image,
  const cc_AcuReadFont &font,
  const cc_AcuReadFont &fontNL,
  const ccAcuReadTuneParams &tuneParams,
  const ccAcuReadRunParams &initial,
  ccAcuReadRunParams &best,
  ccAcuReadResultSet &resultSet);

void tune(ccGreyAcqFifo &fifo,
  const cc2Xform &xform,
  const ccPelRect &rect,
  const cc_AcuReadFont &font,
  const cc_AcuReadFont &fontNL,
```

```
  const ccAcuReadTuneParams &tuneParams,
  const ccAcuReadRunParams &initial,
  ccAcuReadRunParams &best,
  ccAcuReadResultSet &resultSet);
```

- ```
  void tune(const ccPelBuffer_const<c_UInt8> &image,
    const cc_AcuReadFont &font,
    const cc_AcuReadFont &fontNL,
    const ccAcuReadTuneParams &tuneParams,
    const ccAcuReadRunParams &initial,
    ccAcuReadRunParams &best,
    ccAcuReadResultSet &resultSet);
  ```

  Cycles through the preprocessing and size parameters, using the image in *image* and invoking **read()** for each setting. Outputs the best read results and, if **resultSet.found()** is true, the run parameters used to generate them.

  Parameters

image	The pel buffer that contains the image being read.
font	The font of the characters being read.
fontNL	To enable non-linear OCR, specify a **ccAcuReadFont** object initialized with a non-linear font. To disable non-linear OCR, set this parameter to a default-constructed **ccAcuReadFont** object. If *font* is a user-defined font, use a default-constructed **ccAcuReadFont** object for *fontNL*.
tuneParams	The tuning parameters.
initial	The initial run parameters.
best	The run parameters that generated the best read result.
resultSet	The best read result.

  Throws

  ccAcuReadDefs::BadParams
  font is uninitialized,
  the checksum specified in *initial* is not *eNone* and *isVariableLength* is set to true, or
  fontNL is initialized and *font* is initialized to be a user-defined font.

  ccAcuReadDefs::BadImage
  image is unbound.
  ```

- ```
  void tune(ccGreyAcqFifo &fifo,
    const cc2Xform &xform,
    const ccPelRect &rect,
    const cc_AcuReadFont &font,
    const cc_AcuReadFont &fontNL,
    const ccAcuReadTuneParams &tuneParams,
    const ccAcuReadRunParams &initial,
    ccAcuReadRunParams &best,
    ccAcuReadResultSet &resultSet);
  ```

  Cycles through the preprocessing, size, and lighting parameters, using the specified region of interest of an acquired image and invoking **read()** for each setting. Outputs the best read results and, if **resultSet.found()** is true, the run parameters used to generate them.

  Parameters

fifo	An acquisition FIFO.
xform	A coordinate transformation to convert pixel coordinates to client coordinates.
rect	The region of interest.
font	The font of the characters being read.
fontNL	To enable Non-Linear OCR, specify a **ccAcuReadFont** object initialized with a nonlinear font. To disable Non-Linear OCR, set this parameter to a default-constructed **ccAcuReadFont** object.
tuneParams	The tuning parameters.
initial	The initial run parameters.
best	The run parameters that generated the best read result.
resultSet	The best read result.

  Throws

  ccAcuReadDefs::BadParams
  font is uninitialized,
  the checksum specified in *initial* is not *eNone* and *isVariableLength* is set to true, or
  fontNL is initialized and *font* is initialized to be a user-defined font.

---

read

```
void read(ccGreyAcqFifo &fifo,
  const cc2Xform &xform,
  const ccPelRect &rect,
  const cc_AcuReadFont &font,
```
```

```
  const cc_AcuReadFont &fontNL,
  const ccAcuReadRunParams &params,
  ccAcuReadResultSet &resultSet);

void read(const ccPelBuffer_const<c_UInt8> &image,
  const cc_AcuReadFont &font,
  const cc_AcuReadFont &fontNL,
  const ccAcuReadRunParams &params,
  ccAcuReadResultSet &resultSet);
```

- ```
  void read(ccGreyAcqFifo &fifo,
    const cc2Xform &xform,
    const ccPelRect &rect,
    const cc_AcuReadFont &font,
    const cc_AcuReadFont &fontNL,
    const ccAcuReadRunParams &params,
    ccAcuReadResultSet &resultSet);
  ```

  Performs a single read operation on the specified region of interest of an acquired image using the specified parameters. Employs the following sub-functions, in the following order:

  1. Acquisition
  2. Preprocessing
  3. Character segmentation
  4. Character identification
  5. Checksum computation.

  Parameters

fifo	An acquisition FIFO.
xform	A coordinate transformation to convert pixel coordinates to client coordinates.
rect	The region of interest.
font	The font of the characters being read. See *ccAcuReadFont* on page 313.
fontNL	To enable Non-Linear OCR, specify a **ccAcuReadFont** object initialized with a non-linear font. To disable non-linear OCR, set this parameter to a default-constructed **ccAcuReadFont** object.
params	The run parameters.
resultSet	The read result.
  ```

Throws

ccAcuReadDefs::BadParams
font is uninitialized,
the checksum specified in *initial* is not *eNone* and *isVariableLength* is set to true, or
fontNL is initialized and *font* is initialized to be a user-defined font.

- ```
  void read(const ccPelBuffer_const<c_UInt8> &image,
    const cc_AcuReadFont &font,
    const cc_AcuReadFont &fontNL,
    const ccAcuReadRunParams &params,
    ccAcuReadResultSet &resultSet);
  ```

Performs a single read operation on the specified image using the specified parameters. Employs the following sub-functions, in the following order:

1. Preprocessing, using the passed-in image
2. Character segmentation
3. Character identification
4. Checksum computation.

Parameters

image — The pel buffer that contains the image being read.

font — The font of the characters being read. See *ccAcuReadFont* on page 313.

fontNL — To enable non-linear OCR, specify a **ccAcuReadFont** object initialized with a non-linear font. To disable non-linear OCR, set this parameter to a default-constructed **ccAcuReadFont** object.

params — The run parameters.

resultSet — The read result.

Throws

ccAcuReadDefs::BadParams
font is uninitialized,
the checksum specified in *params* is not *eNone* and *isVariableLength* is set to true, or
fontNL is initialized and *font* is initialized to be a user-defined font.

ccAcuReadDefs::BadImage
If *image* is unbound.
  ```

update

```
virtual bool update(const ccAcuReadRunParams &current,
  const ccAcuReadRunParams &best,
  const ccAcuReadResultSet &currentRes,
  const ccAcuReadResultSet &bestRes);
```

Provides a hook that lets you update intermediate **tune()** results. The tuning process continues as long as this function returns true. The default **update()** function always returns true.

Parameters

current	The current acquisition parameters.
best	The best acquisition parameters.
currentRes	The result of the current read operation.
bestRes	The results of the best read operation so far.

userPreprocessAcquiredImage

```
virtual bool userPreprocessAcquiredImage(
  ccPelBuffer<c_UInt8>& src, const ccPelRect& rect,
  ccPelBuffer<c_UInt8>& dst);
```

Allows preprocessing of the acquired image in a custom-defined manner during **tune()** and **read()**. The tune or read process continues as long as this function returns true.

The default implementation assigns the source pel buffer to the destination pel buffer, and sets the window of the destination pel buffer to *rect.* The OCR tool uses the window and client coordinate transform of the destination pel buffer for subsequent processing. You can override the default behavior in a derived class of your own to define other custom preprocessing transformations.

Unless a throw occurs, the default implementation is always to return true. You can override this behavior in a derived class of your own to return false, and abort the **tune()** or **read()** operation in progress, if your custom image preprocessing is unsuccessful. This has nothing to do with the image preprocessing that the OCR tool applies internally, which happens only after this function returns true.

Parameters

src	The acquired image. This pel buffer must be bound.
rect	The region of interest within *src*.
dst	The destination pel buffer. The OCR tool uses the window and client coordinate transform returned in this pel buffer for subsequent processing.

Throws

ccPel::BadWindow

src is not bound, either the **width()** or **height()** of *rect* is not positive, or *rect* is not entirely contained within the window of *src*.

Notes

The default implementation throws under the conditions listed above. You can override this behavior in a derived class of your own.

ccAcuRead

ccAcuReadDefs

```
#include <ch_cvl/acuread.h>

class ccAcuReadDefs;
```

A name space that holds enumerations and constants used with the acuRead classes.

Enumerations

Checksum

```
enum Checksum;
```

Values for the checksum parameter.

Value	Meaning
eNone	No checksum.
eSemi	SEMI checksum.
eBC412	BC412 checksum.
eIBM412	IBM412 checksum.
eVirtual	Virtual checksum.

Fields

```
enum Fields;
```

Convenience enumerations for identifying potential characters in a field.

Value	Meaning
eLastField	Termination of field map.
eAlpha	Uppercase alphabetic character (A through Z).
eNumeric	Numeric character (0 through 9).
eAlphanumeric	Uppercase alphabetic or numeric character.
eDash	A dash or minus (–).
ePeriod	A period or decimal point (.).
eSpace	A space ( ).
eSemiAlpha	A SEMI checksum character (0 through 7 and A through H).

ccAcuReadDefs

Color

```
enum Color;
```

Flags describing character and background colors.

Value	Meaning
eBlack	Black characters on white background.
eWhite	White characters on black background.
eLightInverse	Character color is the inverse of the light mode. This option is for use during light tuning. It allows the character color to change when tuning changes the light mode. It should not be used when the image is stored in a pel buffer.

OcrfFlags

```
enum OcrfFlags;
```

Flags enabling tuning of the filtering modes. These values can be ORed together.

Value	Meaning
eLoPass	Low-pass filter.
eHiPass	High-pass filter.
eTopHat	Top-hat filter.
eClip	Clipping filter.
eAll	All filters combined.

Notes

Use these flags with **ccAcuReadTuneParams** objects only. Do not use them as *preprocessing* options with **ccAcuReadRunParams** objects.

ExitCode

```
enum ExitCode;
```

Codes that indicate how the **ccAcuRead::read()** and **ccAcuRead::tune()** methods terminated.

Notes

ccAcuread::read() only terminates with *eNormal*. **ccAcuRead::tune()** may terminate with any of the values.

Value	Meaning
eNormal = 0	Normal termination.
eValidLimit	The required number of successful reads occurred.
eTimeToValid	The maximum allowed time limit expired.
eScoreLimit	The minimum time limit expired and the score did not exceed the score threshold.

Constants

The following constants are used with the optical character recognition classes:

Value	Meaning
kMaxStringLength = 30	The maximum number of actual characters in a string being read not including the end of field map position.
kMaxFieldStringLength = 39	The maximum length of a field string. (A field string lists the character(s) expected at a specific position.) With field strings that contain multiple special characters, the maximum length is 39 not including the terminating NULL character.

ccAcuReadDefs

ccAcuReadFont

```
#include <ch_cvl/acurdfnt.h>

class ccAcuReadFont: public cc_AcuReadFont;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

ccAcuReadFont encapsulates a font used by the acuRead tool. You populate the object by calling **ccAcuReadFont::access()** and passing it a predefined font name or an **ccOCAlphabetPtrh** object that contains a font set definition.

Constructors/Destructors

ccAcuReadFont

```
ccAcuReadFont();

virtual ~ccAcuReadFont();
```

- `ccAcuReadFont();`

  Default constructor. Initializes the object, but does not provide access to any font.

- `virtual ~ccAcuReadFont();`

Operators

operator==

```
bool operator==(const ccAcuReadFont& rhs) const;
```

Returns true if the font in this object is identical to the font in *rhs*. Returns false otherwise.

Public Member Functions

access

```
virtual bool access(
  const ccCvlString & fontName);

virtual bool access(
  const ccCvlString & fontBaseName,
  bool enableNonLinear);

bool access(
  const ccOCAlphabetPtrh_const & alphabet);
```

- ```
  virtual bool access(
    const ccCvlString & fontName);
  ```

  Searches the list of predefined fonts and provides access to the first such font whose name matches *fontName*. The predefined fonts currently supported are semi_nrm, semi_nln, ocra_nrm, ocra_nln, trip_nrm, trip_nln, cano_nrm and cano_nln.

  Returns true if successful and returns false if unable to find the specified name.

  Parameters

  fontName The font name.

  Notes

  fontName should contain the suffix nrm or nln specifying whether the nonlinear mode or normal mode is selected. For example semi_nrm and semi_nln.

- ```
  virtual bool access(
    const ccCvlString & fontBaseName,
    bool enableNonLinear);
  ```

  You specify the font name to access and the nonlinear or normal mode. The function then searches the list of predefined fonts and provides access to the first such font whose name matches *fontBaseName* with the specified mode enabled.

  Returns true if successful and false if unable to find the specified font name.

  Parameters

  fontBaseName The name of the font you wish to access.

  enableNonLinear
  True for nonlinear mode, and false for normal mode.

  Notes

  fontBaseName should not contain the extension *nrm* or *nln*. It should be one of *semi*, *ocra*, *trip* or *cano*.

- ```
  bool access(
    const ccOCAlphabetPtrh_const & alphabet);
  ```

  Creates a user-defined OCR font from an **ccOCAlphabet** object.

  Returns true if successful. If *alphabet* is invalid, an exception is thrown.

  Parameters

  alphabet An OC alphabet.

  Notes

  Fonts can be either fixed width or proportional width.

  Nonlinear mode is not supported for user-defined fonts. A user-defined font cannot be used as a nonlinear font in a read or tune operation.

  Requirements:

  alphabet must contain no more than 39 characters. Only the characters with the following ASCII codes are allowed: 32 (space), 45 (dash), 46 (period), 48 through 57 (digits), 65 through 90 (uppercase letters).

  The key used must be the ASCII code of the character.

  The character bitmap must be binary. That is, the bitmap must contain only two distinct values. The two values can be anything in the range of 0 through 255. The lower of the two values specifies the background and the higher value specifies the foreground.

  All character bitmaps in the alphabet must have the same height.

  A blank character must be specified with type **eBlank** only. If the alphabet contains a space character, it must be specified with type **eBlank** and have the value 32 as its key.

  Masking is not allowed.

  The area of each character must be less than or equal to 1024 pixels.

  There must be no more than one representation of any character.

  The client transform of all character pel buffers must be the identity transform.

  The string returned by **alphabet->name()** must be less than or equal to 31 characters (not including the NULL terminator).

  Throws

  ccAcuReadDefs::BadParams

  Any of the above requirements are not met.
  ```

ccAcuReadFont

fontName

```
virtual ccCvlString fontName() const;
```

Returns an ANSI string containing the name of the font being accessed. If no font is accessed, a null string is returned,

isAccessed

```
bool isAccessed() const;
```

Returns true if the most recent call to the **access()** method successfully populated the object with font information. Returns false if the call failed, or if **access()** has not yet been called.

isUserDefined

```
bool isUserDefined() const;
```

Returns true if the this font was created from an **ccOCAlphabet** and returns false otherwise.

ccAcuReadResult

```
#include <ch_cvl/acuread.h>

class ccAcuReadResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class holds the results from reading an individual character. You access instances of this class as part of a result set returned by **ccAcuRead::tune()** or **ccAcuRead::read()**. See **ccAcuReadResultSet** on page 321 of the *CVL Class Reference*.

Note You should not construct a **ccAcuReadResult** object directly.

Constructors/Destructors

ccAcuReadResult

```
ccAcuReadResult();
```

Creates a character result. Instances of this class are created automatically as part of a result set.

Operators

operator==

```
bool operator== (const ccAcuReadResult& that) const;
```

Returns true if this object equals *that*. Returns false otherwise.

Two **ccAcuReadResult** objects are considered equal if their **character()**, **location()**, **score()**, and **found()** values are equal. Doubles are compared using **cfRealEq()** with 1.e-8 as the tolerance.

Parameters

that The other **ccAcuReadResult** object.

Public Member Functions

found

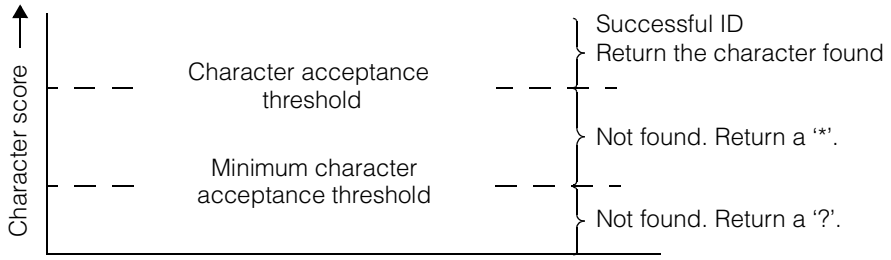
```
bool found() const;
```

Returns true if the character's score was above the *character acceptance threshold*. Returns false otherwise. See the following description.

character

```
c_Int32 character() const;
```

Returns the ASCII code of the found character. If the character was not found but the character score was above the *minimum character acceptance threshold*, it returns an asterisk '*'. If the character score was below the *minimum character acceptance threshold* it returns a question mark '?'. See the following diagram:



You specify an acceptance threshold for the entire string by calling **ccAcuReadRunParams::accept()**. acuRead derives the character acceptance threshold and the minimum character acceptance threshold from this value.

location

```
cc2Vect location() const;
```

Returns the coordinates of identified character in the client coordinate system.

Notes

The returned coordinates are invalid for prefix characters.

score

```
double score() const;
```

Returns the score of the identified character in the range 0.0 through 1.0. A value of 0.0 is always returned for force fitted fields. The value returned for characters that the OCR actually reads, as opposed to those that are force fitted, is a measure of how closely the actual character image data match the identified character. Scores returned for the special characters " " (space), "-" (dash), and "." (period) do not contribute to the overall string score.

ccAcuReadResult

ccAcuReadResultSet

```
#include <ch_cvl/acuread.h>

class ccAcuReadResultSet : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class holds the results of reading a string. In addition to the results for the string as a whole, the result set also includes a vector of results for the individual characters. The functions **ccAcuRead::tune()** and **ccAcuRead::read()** take parameters of type **ccAcuReadResultSet** and fill them with the appropriate results.

Constructors/Destructors

ccAcuReadResultSet

```
ccAcuReadResultSet();
```

Creates a result set. Default-constructed instances of this class can be passed to **ccAcuRead::tune()** and to **ccAcuRead::read()**.

Operators

operator==

```
bool operator== (const ccAcuReadResultSet& that) const;
```

Returns true if this result set is the same as another result set. Two result sets are considered equal if the following attributes are the same: string, angle, checksum, score, acquisition count, and individual character results.

Parameters

that	The other result set.

Public Member Functions

found

```
bool found() const;
```

Returns true if the character string is found. This function returns false if the score for this result is less than the accept threshold *or* if no checksum was valid (and you had enabled at least one checksum).

Notes

Always use **found()** to determine whether or not a read or tune was successful. Do not compare the value of **score()** with an accept threshold to determine success or failure.

readString

```
ccCvlString readString() const;
```

Returns the identified string.

The characters '*' or '?' are placed in positions in the string where incorrect matches were determined during the read or tune operation. See the **acuReadResult::character()** reference material for an explanation of the "*" and "?" characters.

angle

```
ccDegree angle() const;
```

Returns the angle of the identified string in the client coordinate system of the run-time image.

score

```
double score() const;
```

Returns the string score, in the range 0.0 through 1.0.

checksum1Valid

```
bool checksum1Valid() const;
```

Returns true if only one checksum was enabled (either a standard checksum or the Virtual Checksum) and that checksum passed successfully; or if two checksums were enabled (a standard checksum and Virtual Checksum), but only one of them passed.

checksum2Valid

```
bool checksum2Valid() const;
```

Returns true if a standard checksum (SEMI, BC412, or IBM412) was specified with Non-Linear OCR enabled (thereby enabling both the standard checksum and the Virtual Checksum), and both checksums passed.

acquireCount

```
c_Int32 acquireCount() const;
```

Returns the number of acquisitions completed.

result

```
const cmStd vector<ccAcuReadResult> &result() const;
```

Returns a vector of **ccAcuReadResult** objects that represent the results for each individual character. See **ccAcuReadResult** on page 317.

Notes

When running with a fixed string length the size of this vector is set by where you place the *ccAcuReadDefs::eLastField* in the **ccAcuReadRunParams** object. See **ccAcuReadRunParams::fieldString()**.

You set fixed string length mode by setting **isVariableLength()** = false in the **ccAcuReadRunParams** object passed to **ccAcuRead::tune()** or **ccAcuRead::read()**,

time

```
double time() const;
```

Returns the number of seconds the read operation took.

exitCode

```
ccAcuReadDefs::ExitCode exitCode() const;
```

When reading, this function always returns *ccAcuReadDefs::eNormal*.

When tuning it returns one of the following codes indicating how the tuning operation terminated.

ccAcuReadDefs::eNormal
ccAcuReadDefs::eValidLimit
ccAcuReadDefs::eTimeToValid
ccAcuReadDefs::eScoreLimit

runParams

```
const ccAcuReadRunParams &runParams() const;
```

Returns the **ccAcuReadRunParams** used to obtain this result.

nonlinearMode

```
bool nonlinearMode() const;
```

Returns true if nonlinear OCR was used to obtain this result, false otherwise.

ccAcuReadResultSet

ccAcuReadRunParams

```
#include <ch_cvl/acuread.h>

class ccAcuReadRunParams : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class lets you specify parameters used to perform optical character recognition. After you have set up a **ccAcuReadRunParams** object, you pass it to **ccAcuRead::read()**, which performs the reading operation. See **ccAcuRead** on page 301 of the *CVL Class Reference*.

You can set up the reading parameters yourself, or you can use **ccAcuRead::tune()**, which iterates over several possibilities to create a set of parameters for you.

For more detailed information about OCR, reading, and tuning, see the chapter *acuRead* in the *CVL Vision Tools Manual*.

Constructors/Destructors

ccAcuReadRunParams

```
ccAcuReadRunParams();

ccAcuReadRunParams(double accept,
  double acceptNL,
  c_Int32 lineError,
  c_Int32 spaceError,
  c_Int32 charHeight,
  c_Int32 charWidth,
  c_Int32 checksum,
  c_Int32 color,
  c_Int32 preprocessing,
  double lightPower,
  double brightFieldPowerRatio,
```

```
  const ccCvlString & prefix,
  bool isVariableLength = false,
  c_Int32 refine = 3);

ccAcuReadRunParams(const ccAcuReadRunParams &src);

~ccAcuReadRunParams();
```

- `ccAcuReadRunParams();`

  Creates a default parameters object with the following initial values:

Parameter	Initial Value
accept	0.40
acceptNL	0.7
lineError	2
spaceError	4
charHeight	28
charWidth	14
checksum	*ccAcuReadDefs::eNone*
color	*ccAcuReadDefs::eLightInverse*
preprocessing	0
lightPower	0
brightFieldPowerRatio	0
prefix	““
isVariableLength	false
refine	3

  The field map contains the string “0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ” for positions 0 through *kMaxStringLength* - 1, and *eLastField* for position *kMaxStringLength*.

-
```
ccAcuReadRunParams(
  double accept,
  double acceptNL,
```

```
c_Int32 lineError,
c_Int32 spaceError,
c_Int32 charHeight,
c_Int32 charWidth,
c_Int32 checksum,
c_Int32 color,
c_Int32 preprocessing,
double lightPower,
double brightFieldPowerRatio,
const ccCvlString & prefix,
bool isVariableLength = false,
c_Int32 refine = 3);
```

Creates a parameters object with the specified values.

Parameters

accept	The acceptance threshold for standard OCR. Valid range, 0.0 through 1.0.
acceptNL	The acceptance threshold in nonlinear mode. Valid range, 0.0 through 1.0.
lineError	The line error tolerance in pixels. Valid range, 0 through 20.
spaceError	The space error tolerance in pixels. Valid range, 0 through 20.
charHeight	The initial character height in pixels. Valid range, 8 through 128.
charWidth	The initial character width in pixels. Valid range, 8 through 64.
checksum	The checksum method. Must be one of the following values: *ccAcuReadDefs::eNone* *ccAcuReadDefs::eSemi* *ccAcuReadDefs::eVirtual* For details, see *checksum* on page 336.
color	The character color. Must be one of: *ccAcuReadDefs::eBlack* *ccAcuReadDefs::eWhite* *ccAcuReadDefs::eLightInverse* For details, see *Color* on page 310.
preprocessing	The preprocessing filters to apply. Must be one of the following values:

0: 1D TopHat
1: 1D TopHat + Low Pass
2: 1D High Pass
3: 1D High Pass + Low Pass
4: 2D Top Hat
5: 2D Top Hat + Low Pass
6: 2D High Pass
7: 2D High Pass + Low Pass
8: Clip + 1D TopHat
9: Clip + 1D TopHat + Low Pass
10: Clip + 1D High Pass
11: Clip + 1D High Pass + Low Pass
12: Clip + 2D Top Hat
13: Clip + 2D Top Hat + Low Pass
14: Clip + 2D High Pass
15: Clip + 2D High Pass + Low Pass

lightPower The light power level to be used. Valid range is 0.0 through 1.0.

brightFieldPowerRatio
The bright field power ratio specifies the fraction of the total light power allocated to the bright field.

Also see the description under *brightFieldPowerRatio* on page 338.

prefix The prefix string prepended to each read string before calculating the checksum.

isVariableLength
Set to true if the length of the string(s) to be read is not known. Set to false otherwise.

Also see description under *isVariableLength* on page 339.

refine The refinement strategy. Must be one of the following values:

0: no refinement
1: scale refinement
2: field string compliance
3: scale refinement and field string compliance (the default)

Also see notes under *refine* on page 339.

ccAcuReadDefs::BadParams
A parameter value was out of range.

- `ccAcuReadRunParams(const ccAcuReadRunParams &src);`

  Copy constructor.

  Parameters

  src The source **ccAcuReadRunParams** object.

- `~ccAcuReadRunParams();`

  Destroys a **ccAcuReadRunParams** object.

Operators

operator== `bool operator== (const ccAcuReadRunParams& that) const;`

Returns true if the attribute values in the current run parameters object are equal to those in the other.

Parameters

that The other **ccAcuReadRunParams** object.

Public Member Functions

accept

```
double accept() const;
void accept(double val);
```

- `double accept() const;`

  Returns the acceptance threshold value. This is the string acceptance threshold used by the normal mode OCR. A string must achieve at least this score to be reported as found.

  The acceptance threshold is the minimum score that a character needs to achieve to be considered a valid character.

- `void accept(double val);`

  Sets the acceptance threshold value.

  Parameters

  val The acceptance threshold. Must be a value in the range 0.0 through 1.0.

Throws

ccAcuReadDefs::BadParams
val was out of range.

acceptNL

```
double acceptNL() const;
void acceptNL(double val);
```

- `double acceptNL() const;`

  Returns the acceptance threshold for nonlinear mode.

- `void acceptNL(double val);`

  Sets the acceptance threshold for nonlinear mode.

  Parameters

  val The acceptance threshold. Must be a value in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

lineError

```
c_Int32 lineError() const;
void lineError(c_Int32 val);
```

- `c_Int32 lineError() const;`

  Returns the line error tolerance. The line error tolerance is the number of pixels that a character can be from an ideal baseline and still be considered a match.

- `void lineError(c_Int32 val);`

  Sets the line error tolerance.

  Parameters

  val The line error tolerance in pixels. Must be a value in the range 0 through 20.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

spaceError

```
c_Int32 spaceError() const;

void spaceError(c_Int32 val);
```

- `c_Int32 spaceError() const;`

  Returns the space error tolerance. The space error tolerance is the number of pixels that a character can deviate left or right from an ideal center line that spaces all of the characters equally.

- `void spaceError(c_Int32 val);`

  Sets the space error tolerance.

  Parameters

  val The space error tolerance in pixels. Must be a value in the range 0 through 20.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

fieldString

```
ccCvlString fieldString(c_Int32 position) const;

void fieldString(c_Int32 position,
 const ccCvlString &str);

void fieldString(c_Int32 position,
 ccAcuReadDefs::FieldType type);
```

- `ccCvlString fieldString(c_Int32 position) const;`

  Returns the string that lists the characters that are valid at the specified position in the string to be read.

  Parameters

  position The character position. Must be in the range 0 through *kMaxStringLength*.

  Throws

  ccAcuReadDefs::BadParams
  position was out of range.

- `void fieldString(c_Int32 position, const ccCvlString &str);`

  Specifies the characters that are valid at the specified *position* in the string to be read. "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" is the default field string for character positions 0 through *KMaxStringLength* - 1.

  Parameters

position	The character position. Must be in the range 0 through *kMaxStringLength*.
str	String containing the acceptable characters at the specified field position. For positions 0 through *kMaxStringLength* - 1, the string can either be empty (if this field is intended to be the end of the string to be read) or contain any combination of alphanumeric characters 'A' through 'Z', '0' through '9', or special characters '–' (dash), ' ' (space), or '.' (period), without repetition. For position *kMaxStringLength*, the string must be empty.

  Example

  The following code sets the field string at field position 3 to accept numerics and special characters:

  ```
  {...
   ccAcuReadRunParams rp;
   rp.fieldString(3, "0123456 -.");
   ...
  }
  ```

  Throws

  ccAcuReadDefs::BadParams

  position was out of range, *str* included an invalid or duplicate character, a non-empty *str* is specified for position kMaxStringLength, or the length of *str* was greater than *ccAcuReadDefs::kMaxFieldStringLength–1*.

- `void fieldString(c_Int32 position, ccAcuReadDefs::FieldType type);`

  Sets the field string at the specified *position* to a predefined *fieldType* value.

  Parameters

position	The character position to be set. Must be in the range 0 through *kMaxStringLength*.
type	The type of field that is valid at the specified position.

For positions 0 through *kMaxStringLength* - 1, the following are valid field types:
ccAcuReadDefs::eAlpha ('A' through 'Z')
ccAcuReadDefs::eNumeric, ('0' through '9')
ccAcuReadDefs::eAlphanumeric, (any alphanumeric)
ccAcuReadDefs::eDash, ('-' character)
ccAcuReadDefs::ePeriod, ('.' character)
ccAcuReadDefs::eSpace, (' ' character)
ccAcuReadDefs::eSemiAlpha (Semi checksum character '0' through '7' or 'A' through 'H')

For position *kMaxStringLength*, the only valid field type is *ccAcuReadDefs::eLastField*.

See *Fields* on page 309.

Throws

ccAcuReadDefs::BadParams
position was out of range, *type* was an unknown or invalid type, or a type other than *ccAcuReadDefs::eLastField* was specified for position *kMaxStringLength*.

charHeight

```
c_Int32 charHeight() const;
void charHeight(c_Int32 val);
```

- `c_Int32 charHeight() const;`

Returns the height of the characters to be read.

- `void charHeight(c_Int32 val);`

Sets the height of the characters to be read. Values larger than the height of the region of interest cause failed reads.

Parameters

val — The expected height, in pixels, of the characters to be read. Values must be in the range 8 through 128.

Notes

If this object is being used to pass the initial parameters to **ccAcuRead::tune()**, it is recommended that *val* be equal to the *heightNominal* value of the **ccAcuReadTuneParams** object being used to pass the tuning parameters.

Throws

ccAcuReadDefs::BadParams
val was out of range.

charWidth

```
c_Int32 charWidth() const;
void charWidth(c_Int32 val);
```

- `c_Int32 charWidth() const;`

  Returns the width of the characters to be read.

- `void charWidth(c_Int32 val);`

  Sets the width of the characters to be read. Values larger than the width of the region of interest cause failed reads.

  Parameters

  val — The expected width, in pixels, of the characters to be read. Values must be in the range 8 through 64.

  Notes

  If this object is being used to pass the initial parameters to **ccAcuRead::tune()**, it is recommended that *val* be equal to the *widthNominal* value of the **ccAcuReadTuneParms** object being used to pass the tuning parameters.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

color

```
ccAcuReadDefs::Color color() const;
void color(ccAcuReadDefs::Color val);
```

- `ccAcuReadDefs::Color color() const;`

  Returns the character color.

- `void color(ccAcuReadDefs::Color val);`

  Sets the character color.

Notes

The default color, *ccAcuReadDefs::eLightInverse*, should not be used with the overloads of **ccAcuRead::read()** and **ccAcuRead::tune()** that examine images stored in pel buffers.

Parameters

val The character color. Must be one of:

ccAcuReadDefs::eBlack
ccAcuReadDefs::eWhite
ccAcuReadDefs::eLightInverse

For details, see *Color* on page 310.

Throws

ccAcuReadDefs::BadParams
val was an unsupported value.

preprocessing

```
c_UInt32 preprocessing() const;

void preprocessing(c_UInt32 val);
```

- `c_UInt32 preprocessing() const;`

Returns the preprocessing options.

- `void preprocessing(c_UInt32 val);`

Sets the preprocessing options to use when reading a string.

Parameters

val The preprocessing options. Must be one of the following values:

0: 1D TopHat
1: 1D TopHat + Low Pass
2: 1D High Pass
3: 1D High Pass + Low Pass
4: 2D Top Hat
5: 2D Top Hat + Low Pass
6: 2D High Pass
7: 2D High Pass + Low Pass
8: Clip + 1D TopHat
9: Clip + 1D TopHat + Low Pass
10: Clip + 1D High Pass
11: Clip + 1D High Pass + Low Pass
12: Clip + 2D Top Hat

13: Clip + 2D Top Hat + Low Pass
14: Clip + 2D High Pass
15: Clip + 2D High Pass + Low Pass

The 1D and 2D versions of the top-hat and high-pass filters enable you to choose between one-dimensional and two-dimensional operations on pixel values. The 2D options are more accurate, but slower than the 1D options.

Notes

If this runparams object was initialized by **ccAcuRead::tune()**, changing the preprocessing options is not recommended.

Throws

ccAcuReadDefs::BadParams
val was an unsupported value.

checksum

```
ccAcuReadDefs::Checksum checksum() const;
void checksum(ccAcuReadDefs::Checksum val);
```

- `ccAcuReadDefs::Checksum checksum() const;`

Returns the checksum method.

- `void checksum(ccAcuReadDefs::Checksum val);`

Sets the checksum method or methods.

Parameters

val The checksum method. Must be one of the following values:

Checksum selected	Interpretation
ccAcuReadDefs::eNone	*eNone*
ccAcuReadDefs::eSemi	*eSemi + eVirtual*
ccAcuReadDefs::eBC412	*eBC412 + eVirtual*
ccAcuReadDefs::eIBM412	*eIBM412 + eVirtual*
ccAcuReadDefs::eVirtual	*eVirtual*

Throws

ccAcuReadDefs::BadParams
val was an unsupported value.

Notes

ccAcuReadDefs::eNone disables all checksums.

ccAcuReadDefs::eVirtual enables Virtual Checksum if Non-Linear OCR is enabled, and is ignored if Non-Linear OCR is not enabled.

ccAcuReadDefs::eSemi enables the SEMI checksum if Non-Linear OCR is disabled, and enables both the SEMI checksum and the Virtual Checksum when Non-Linear OCR is enabled.

eVirtual is ignored when using user-defined fonts.

Checksum is not supported for variable length strings. If the checksum method is set, the **ccAcuRead::tune()** and **ccAcuRead::run()** member functions will throw an exception. Refer to the **ccAcuRead::tune()** and **ccAcuRead::read()** methods for information on enabling Non-Linear OCR.

lightPower

```
double lightPower() const;
void lightPower(double val);
```

- `double lightPower() const;`

Returns the light power value.

- `void lightPower(double val);`

Sets the light power.

Parameters

val The light power. Must be in the range 0.0 through 1.0.

Throws

ccAcuReadDefs::BadParams
val was out of range.

brightFieldPowerRatio

```
double brightFieldPowerRatio() const;
void brightFieldPowerRatio(double val);
```

The bright field power ratio specifies the fraction of the total light power allocated to the bright field. The setting must be in the range 0.0 through 1.0. A value of 0.0 indicates bright field is completely off, and a value of 1.0 indicates bright field is fully on.

This is a tunable parameter.

- `double brightFieldPowerRatio() const;`

  Returns the current bright field power ratio setting.

- `void brightFieldPowerRatio(double val);`

  Sets a new bright field power ratio setting.

  Parameters

  val The new bright field power ratio setting.

  Throws

  ccAcuReadDefs::BadParams
  If *val* is outside the valid range.

prefix

```
ccCvlString prefix() const;
void prefix(const ccCvlString & str);
```

- `ccCvlString prefix() const;`

  Returns the prefix string prepended to each read string before calculating the checksum.

- `void prefix(const ccCvlString & str);`

  Sets the prefix string prepended to each read string before calculating the checksum.

  Parameters

  str The prefix string.

refine

```
c_Int32 refine() const;
void refine(c_Int32 val);
```

The result refinement strategy. Controls the scale refinement and field string compliance as described below:

Value	Scale refinement	Field string compliance
0	Disabled	Disabled
1	Enabled	Disabled
2	Disabled	Enabled
3 (default)	Enabled	Enabled

- `c_Int32 refine() const;`

  Returns the refinement strategy.

- `void refine(c_Int32 val);`

  Sets a new refinement strategy.

  Parameters

  val The new refinement strategy.

  Throws

  ccAcuReadDefs::BadParams
  If *val* is outside the valid range.

  Notes

  Turning off string compliance causes acuRead to ignore any specified field strings and to use the default field string (‘0’ through ‘9’, ‘A’ through ‘Z’) to read all characters.

  This function is included for backward compatibility; its use is not recommended.

isVariableLength

```
bool isVariableLength() const;
void isVariableLength(bool isVarLen);
```

Set *isVariableLength* to true if the length of the string(s) to be read is not known. When set to true, the tool considers the length of the field map as the maximum possible value. Consequently, the read operation can result in a string that is smaller than the field map. If set to false, the resulting string length will always be equal to that of the field map.

- `bool isVariableLength() const;`

  Returns the *isVariableLength* flag; true or false.

- `void isVariableLength(bool isVarLen);`

  Sets the *isVariableLength* flag.

  Parameters

  isVarLen The new flag value, true of false.

  Notes

  Characters at the rightmost and leftmost positions of a string are termed end characters. The tool will not report any end characters whose score is below the accept threshold. These will not affect the overall score of the string. However, non-end (middle) characters with a “?” or a “*” will be reported and will affect the overall score. For information about characters reported as “?” or “*” see the **ccAcuReadResult::character()** reference material.

  It may not be possible to provide certain types of information using the field string API for variable length strings. For example, if a single “-” is expected in a string but its position varies from one string to another, and you do not want the “-” to be a choice for other positions in the field string, there is no way to specify this case with the API.

  Checksum is not supported for variable length strings. If the checksum method is set, the **ccAcuRead::tune()** and **ccAcuRead::run()** member functions will throw an exception. Refer to the **ccAcuRead::tune()** and **ccAcuRead::read()** methods for information.

Deprecated Members

The following function is deprecated and is now maintained for backward compatibility only. For new code development use **brightFieldPowerRatio()**.

darkLevel

```
double darkLevel() const;

void darkLevel(double val);
```

- `double darkLevel() const;`

  Returns the dark level to be used. The range must be between 0.0 and 1.0.

- `void darkLevel(double val);`

  Sets the darkLevel to be used. The range must be between 0.0 and 1.0.

  Parameters

  val Dark level in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

ccAcuReadRunParams

ccAcuReadTuneParams

```
#include <ch_cvl/acuread.h>

class ccAcuReadTuneParams : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class lets you specify the parameters used to tune the lighting, filtering, and character-scaling parameters that affect optical character recognition.

In addition to passing **ccAcuRead::tune()** a **ccAcuReadRunParams** object with initial reading parameters, you pass a **ccAcuReadTuneParams** object to specify how the tuning function should alter the initial parameters as it attempts to find the run parameters that yield the best string score. See **ccAcuRead** on page 301.

For more detailed information about OCR, reading, and tuning, see the chapter *acuRead* in the *CVL Vision Tools Manual*.

Constructors/Destructors

ccAcuReadTuneParams

```
ccAcuReadTuneParams();

ccAcuReadTuneParams(bool enableDark,
  double darkLow,
  double darkHigh,
  double darkStep,
  bool enableBright,
  double brightLow,
  double brightHigh,
  double brightStep,
  bool enableHeight,
  c_Int32 heightNominal,
  c_Int32 heightRange,
  bool enableWidth,
  c_Int32 widthNominal,
  c_Int32 widthRange,
  c_UInt32 enableOcrf,
  c_Int32 validLimit,
  double timeToValid,
```

```
  double scoreLimit,
  bool scoreLimitChecksum,
  double timeToScore);

~ccAcuReadTuneParams();
```

- `ccAcuReadTuneParams();`

  Creates a default tune-parameters object. The initial parameters are set as follows:

Parameter	Value
enableDark	true
darkLow	0.062745
darkHigh	1.0
darkStep	0.12157
enableBright	true
brightLow	0.062745
brightHigh	1.0
brightStep	0.12157
enableHeight	true
heightNominal	28
heightRange	4
enableWidth	true
widthNominal	14
widthRange	2
enableOcrf	*ccAcuReadDefs::eAll*
validLimit	255
timeToValid	30000
scoreLimit	0.10
scoreLimitChecksum	false
timeToScore	30000

- ```
  ccAcuReadTuneParams(bool enableDark,
    double darkLow,
    double darkHigh,
    double darkStep,
    bool enableBright,
    double brightLow,
    double brightHigh,
    double brightStep,
    bool enableHeight,
    c_Int32 heightNominal,
    c_Int32 heightRange,
    bool enableWidth,
    c_Int32 widthNominal,
    c_Int32 widthRange,
    c_UInt32 enableOcrf,
    c_Int32 validLimit,
    double timeToValid,
    double scoreLimit,
    bool scoreLimitChecksum,
    double timeToScore);
  ```

  Parameters

enableDark	True to enable darkfield tuning.
darkLow	The minimum light power for darkfield tuning. Must be in the range 0.0 through 1.0.
darkHigh	The maximum light power for darkfield tuning. Must be in the range 0.0 through 1.0.
darkStep	The darkfield step value. Must be in the range 0.0 through 1.0.
enableBright	True to enable brightfield tuning.
brightLow	The minimum light power for brightfield tuning. Must be in the range 0.0 through 1.0.
brightHigh	The maximum light power for brightfield tuning. Must be in the range 0.0 through 1.0.
brightStep	The brightfield step value. Must be in the range 0.0 through 1.0.
enableHeight	True to enable tuning for character height.
heightNominal	The nominal character height, in pixels. Must be in the range 8 through 128.
heightRange	The character height range, in pixels. Must be in the range 0 through 16.
enableWidth	True to enable tuning of character width.
  ```

widthNominal The nominal character width, in pixels. Must be in the range 8 through 64.

widthRange The character width range, in pixels. Must be in the range 0 through 16.

enableOcrf One of the following values that specify which filters to tune (or zero to disable filter tuning):

ccAcuReadDefs::eLoPass
ccAcuReadDefs::eHiPass
ccAcuReadDefs::eTopHat
ccAcuReadDefs::eClip
ccAcuReadDefs::eAll

validLimit The maximum number of successful reads that can occur without producing a new best score.

timeToValid The maximum number of seconds to spend tuning.

scoreLimit The score threshold for early termination of tuning. Must be in the range 0.0 through 1.0.

scoreLimitChecksum
True terminates tuning if the checksum passes.

timeToScore The maximum time, in seconds, that can elapse before tuning reaches *scoreLimit*.

Throws

ccAcuReadDefs::BadParams
A parameter value was out of range.

- `~ccAcuReadTuneParams();`

Destroys a tune-parameters object.

Operators

operator== `bool operator== (const ccAcuReadTuneParams& that) const;`

Returns true if these tuning parameters are the same as another set. Two tuning parameter sets are considered equal if all of their attributes are the same.

Parameters

that The other tuning parameter set.

Public Member Functions

enableDark

```
bool enableDark() const;
void enableDark(bool val);
```

- `bool enableDark() const;`

  Returns true if darkfield tuning is enabled.

- `void enableDark(bool val);`

  Enables or disables darkfield tuning.

  Parameters

  val — True to enable darkfield tuning, false to disable.

darkLow

```
double darkLow() const;
void darkLow(double val);
```

- `double darkLow() const;`

  Returns the minimum light power for darkfield tuning.

- `void darkLow(double val);`

  Sets the minimum light power for darkfield tuning.

  Parameters

  val — The minimum light power for darkfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

darkHigh

```
double darkHigh() const;
void darkHigh(double val);
```

- `double darkHigh() const;`

  Returns the maximum light power for darkfield tuning.

- `void darkHigh(double val);`

  Sets the maximum light power for darkfield tuning.

  Parameters

  val The maximum light power for darkfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

darkStep

```
double darkStep() const;
void darkStep(double val);
```

- `double darkStep() const;`

  Returns the darkfield step for tuning.

- `void darkStep(double val);`

  Sets the darkfield step for tuning.

  Parameters

  val The darkfield step value. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

enableBright

```
bool enableBright() const;
void enableBright(bool val);
```

- `bool enableBright() const;`

  Returns true if brightfield tuning is enabled.

- `void enableBright(bool val);`

  Enables or disables brightfield tuning.

  Parameters

  val True to enable brightfield tuning, false to disable.

brightLow

```
double brightLow() const;
void brightLow(double val);
```

- `double brightLow() const;`

  Returns the minimum light power for brightfield tuning.

- `void brightLow(double val);`

  Sets the minimum light power for brightfield tuning.

  Parameters

  val The minimum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

brightHigh

```
double brightHigh() const;
void brightHigh(double val);
```

- `double brightHigh() const;`

  Returns the maximum light power for brightfield tuning.

- `void brightHigh(double val);`

  Returns the maximum light power for brightfield tuning.

  Parameters

  val The maximum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

ccAcuReadTuneParams

brightStep

```
double brightStep() const;
void brightStep(double val);
```

- `double brightStep() const;`

  Returns the brightfield step for tuning.

- `void brightStep(double val);`

  Sets the brightfield step for tuning.

  Parameters

  val The brightfield step value. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

enableHeight

```
bool enableHeight() const;
void enableHeight(bool val);
```

- `bool enableHeight() const;`

  Returns true if character height tuning is enabled.

- `void enableHeight(bool val);`

  Enables or disables character height tuning.

  Parameters

  val True to enable character height tuning, false to disable.

heightNominal

```
c_Int32 heightNominal() const;
void heightNominal(c_Int32 val);
```

- `c_Int32 heightNominal() const;`

  Returns the nominal (initial) character height.

- `void heightNominal(c_Int32 val);`

  Sets the nominal (initial) character height.

  Parameters

  val — The nominal character height, in pixels. Must be in the range 8 through 128. It is recommended that *val* be equal to the *charHeight* value of the **ccAcuReadRunParams** object being used to pass the initial parameters to **ccAcuRead::tune()**.

  Throws

  ccAcuReadDefs::BadParams — *val* was out of range.

heightRange

`c_Int32 heightRange() const;`

`void heightRange(c_Int32 val);`

- `c_Int32 heightRange() const;`

  Returns the tolerance that controls the range of character heights tested during tuning. The range is twice this value, which is a plus-and-minus tolerance applied to the nominal height.

- `void heightRange(c_Int32 val);`

  Sets the tolerance that controls the range of character heights tested during tuning. The range is twice this value, which is a plus-and-minus tolerance applied to the nominal height.

  Parameters

  val — The character height tolerance, in pixels. Must be in the range 0 through 16.

  Throws

  ccAcuReadDefs::BadParams — *val* was out of range.

  Notes

  During tuning, acuRead sometimes tests values outside the range you specify, with the result that the character height returned after tuning may fall outside the range (*heightNominal* – *heightRange*, *heightNominal* + *heightRange*).

ccAcuReadTuneParams

enableWidth

```
bool enableWidth() const;
void enableWidth(bool val);
```

- `bool enableWidth() const;`

  Returns true if character width tuning is enabled.

- `void enableWidth(bool val);`

  Enables or disables character width tuning.

  Parameters

  val — True to enable character width tuning, false to disable.

widthNominal

```
c_Int32 widthNominal() const;
void widthNominal(c_Int32 val);
```

- `c_Int32 widthNominal() const;`

  Returns the nominal (initial) character width.

- `void widthNominal(c_Int32 val);`

  Sets the nominal (initial) character width.

  Parameters

  val — The nominal character width, in pixels. Must be in the range 8 through 64. It is recommended that *val* be equal to the *charWidth* value of the **ccAcuReadRunParams** object being used to pass the initial parameters to **ccAcuRead::tune()**.

  Throws

  ccAcuReadDefs::BadParams — *val* was out of range.

widthRange

```
c_Int32 widthRange() const;

void widthRange(c_Int32 val);
```

- `c_Int32 widthRange() const;`

  Returns the tolerance that controls the range of character widths tested during tuning. The range is twice this value, which is a plus-and-minus tolerance applied to the nominal width.

- `void widthRange(c_Int32 val);`

  Sets the tolerance that controls the range of character widths tested during tuning. The range is twice this value, which is a plus-and-minus tolerance applied to the nominal width.

  Parameters

  val — The character width tolerance, in pixels. Must be in the range 0 through 16.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

  Notes

  During tuning, acuRead sometimes tests values outside the range you specify, with the result that the character width returned after tuning may fall outside the range (*widthNominal* – *widthRange*, *widthNominal* + *widthRange*).

enableOcrf

```
c_UInt32 enableOcrf() const;

void enableOcrf(c_UInt32 code);
```

- `c_UInt32 enableOcrf() const;`

  Returns zero if filter tuning is disabled or, when filter tuning is enabled, a value formed by ORing together one or more of the following values:

  ccAcuReadDefs::eLoPass
  ccAcuReadDefs::eHiPass
  ccAcuReadDefs::eTopHat
  ccAcuReadDefs::eClip
  ccAcuReadDefs::eAll

  The returned value indicates which filters will be considered during filter tuning.

- `void enableOcrf(c_UInt32 code);`

  Specifies which filters will be considered during filter tuning.

  Parameters

  code The preprocessing options. Use zero to disable filter tuning. Use any of the following values, which can be ORed together, to enable filter tuning:

  ccAcuReadDefs::eLoPass
  ccAcuReadDefs::eHiPass
  ccAcuReadDefs::eTopHat
  ccAcuReadDefs::eClip
  ccAcuReadDefs::eAll

  For details, see *OcrfFlags* on page 310.

  Throws

  ccAcuReadDefs::BadParams
  If *code* is an invalid value.

  Notes

  Either *ccAcuReadDefs::eTopHat* or *ccAcuReadDefs::eHiPass* must be selected for tuning to take place. If one of these filters is not chosen, filter tuning is not performed.

validLimit

```
c_Int32 validLimit() const;
void validLimit(c_Int32 val);
```

- `c_Int32 validLimit() const;`

  Returns the maximum number of successful reads that can be performed during tuning without producing a new best score.

- `void validLimit(c_Int32 val);`

  Sets the maximum number of successful reads that can occur during tuning without producing a new best score.

  Parameters

  val The maximum number of successful reads. Must be in the range 0 through 255.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

timeToValid

```
double timeToValid() const;
void timeToValid(double val);
```

- `double timeToValid() const;`

  Returns the maximum number of seconds to spend tuning.

- `void timeToValid(double val);`

  Sets the maximum number of seconds to spend tuning.

  Parameters

  val The maximum number of seconds to spend tuning. Must be nonnegative

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

scoreLimit

```
double scoreLimit() const;
void scoreLimit(double val);
```

- `double scoreLimit() const;`

  Returns the string score that tuning must reach within **timeToScore()** seconds in order for tuning to continue.

- `void scoreLimit(double val);`

  Sets the string score that tuning must reach within **timeToScore()** seconds to continue.

  If **scoreLimitChecksum()** is true, the string being read will also need to pass its checksum for tuning to continue beyond **timeToScore()** seconds.

  Parameters

  val The score threshold for early termination of tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuReadDefs::BadParams
  val was out of range.

scoreLimitChecksum

```
bool scoreLimitChecksum() const;
void scoreLimitChecksum(bool val);
```

- `bool scoreLimitChecksum() const;`

  Returns true if the tuning operation considers the checksum in addition to the string score to determine whether tuning should continue beyond **timeToScore()** seconds.

- `void scoreLimitChecksum(bool val);`

  Specifies whether or not the string must pass its checksum within **timeToScore()** seconds for tuning to continue.

  Parameters

  val — True to use the checksum, false to exclude the checksum.

  Notes

  If no checksum is enabled by the *initial* run parameters you pass to **ccAcuRead::tune()**, setting **scoreLimitChecksum()** true has no effect.

timeToScore

```
double timeToScore() const;
void timeToScore(double val);
```

- `double timeToScore() const;`

  Returns the maximum number of seconds that tuning can continue before the string reaches **scoreLimit()** and, if **scoreLimitChecksum()** is true, passes its checksum.

- `void timeToScore(double val);`

  Sets the maximum number of seconds that tuning can continue before the string reaches **scoreLimit()** and, if **scoreLimitChecksum()** is true, passes its checksum.

  Parameters

  val — The maximum number of seconds to tune without reaching the desired string score and, if required, passing the checksum. Must be nonnegative.

  Throws

  ccAcuReadDefs::BadParams — *val* was out of range.

ccAcuSymbolDataMatrixDefs

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolDataMatrixDefs: public ccAcuSymbolDefs;
```

A name space that holds enumerations and constants used with the Symbol tool classes.

Enumerations

ECCType

```
enum ECCType;
```

Values for the ECCType parameter:

Value	Meaning
eECC0	ECC level 0
eECC050	ECC level 50
eECC080	ECC level 80
eECC100	ECC level 100
eECC140	ECC level 140
eECC200	ECC level 200
kDefaultECC	*eECC200*

LearnFlags

```
enum LearnFlags;
```

Setting the following LearnFlags determines which parameters the tool learns:

Value	Meaning
eNone = 0	Tool learns no parameters.
eECC = 1	Tool learns ECC level.
eGrid = 2	Tool learns Grid size.
eNominalGrid = 4	Tool learns nominal grid.
ePolarity = 8	Tool learns polarity.
eModel = 16	Turns optimization on.

Value	Meaning
eAll = 31	Tool learns all learning parameters.
kDefaultLearn = eAll	Default is for the tool to learn all learning parameters.

kMaxDataMatrixModelSize

```
enum kMaxDataMatrixModelSize;
```

Maximum size for a Data Matrix symbol:

Value	Meaning
kMaxDataMatrixModelSize = 48	The maximum supported symbol size is 48.

Polarity

```
enum Polarity;
```

Values for the Polarity parameter:

Value	Meaning
eDarkOnLight	Dark symbol on a light background
eLightOnDark	Light symbol on a dark background

ccAcuSymbolDataMatrixLearnParams

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolDataMatrixLearnParams:
  public ccAcuSymbolLearnParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The class **ccAcuSymbolDataMatrixLearnParams** supplies the parameters that the Symbol tool uses to create a Data Matrix symbol model. The tool can acquire values for the parameters in this class in the following ways:

- You can set the Learn parameters to their default values by using the default constructor.
- You can specify values for specific Learn parameters using the overload constructor.
- You can set values for the Learn parameters using the setters of the class.
- You can have the tool learn the parameters specified by **learnFlags**. For those parameters not learned by the tool, you use this class to specify the values.

Constructors/Destructors

ccAcuSymbolDataMatrixLearnParams

```
ccAcuSymbolDataMatrixLearnParams();

virtual ~ccAcuSymbolDataMatrixLearnParams();

ccAcuSymbolDataMatrixLearnParams(
   const ccAffineRectangle& nominalGrid,
   c_UInt8 nrows,
   c_UInt8 ncols,
   ccAcuSymbolDataMatrixDefs::ECCType ecc =
      ccAcuSymbolDataMatrixDefs::kDefaultECC,
   ccAcuSymbolDataMatrixDefs::Polarity symbolPolarity =
   ccAcuSymbolDataMatrixDefs::eDarkOnLight,
   ccRadian angle = ccRadian(0.0),
```

```
   double scale = 1.0,
   bool aimFlag = true,
   bool mirror = false);
```

- `ccAcuSymbolDataMatrixLearnParams();`

  Default constructor. The members are initialized to their default values, which are:

Parameter	Value
nRows	10
nCols	10
nominalGrid	Points at (0, 10), (10, 10), (0, 0)
eccType	*ccAcuSymbolDataMatrixDefs::kDefaultECC*
symbolPolarity	*ccAcuSymbolDataMatrixDefs::eDarkOnLight*
aimFlag	True

  Notes

  The nominal grid as well as the number of rows and columns are set to match the smallest symbol size (10 x 10) allowed for an AIM-compliant Data Matrix symbol (version ECC 200). The transform of the nominal grid is set to identity.

- `virtual ~ccAcuSymbolDataMatrixLearnParams();`

  Destroys instances of this class.

- ```
  ccAcuSymbolDataMatrixLearnParams(
     const ccAffineRectangle& nominalGrid,
     c_UInt8 nrows,
     c_UInt8 ncols,
     ccAcuSymbolDataMatrixDefs::ECCType ecc =
        ccAcuSymbolDataMatrixDefs::kDefaultECC,
     ccAcuSymbolDataMatrixDefs::Polarity symbolPolarity =
     ccAcuSymbolDataMatrixDefs::eDarkOnLight,
     ccRadian angle = ccRadian(0.0),
     double scale = 1.0,
     bool aimFlag = true,
     bool mirror = false);
  ```

  Constructor overload with specified values.
  ```

Parameters

nominalGrid The affine rectangle that encloses the symbol.

nrows The number of rows in the symbol. For AIM-compliant symbols, the number of rows must be the same as the number of columns for the symbol grid for all versions except *eECC200*.

For AIM-compliant square symbols, *nrows* should be:

Odd and in the range 9 through *kMaxDataMatrixModelSize* if the ECC type specified is *eECC0-eECC140*.

Even and in the range 10 through *kMaxDataMatrixModelSize* if the version specified is *eECC200*.

For AIM-compliant rectangular symbols (version ECC 200 only), *nrows* must be one of the following combinations:

nrows	ncols
8	18
8	32
12	26
12	36
16	36
16	48

ncols The number of columns in the symbol. For AIM-compliant symbols, the number of rows must be the same as the number of columns for the symbol grid for all versions except *eECC200*.

For AIM-compliant square symbols, *ncols* should be:

Odd and in the range 9 through *kMaxDataMatrixModelSize* if the ECC type specified is *eECC0-eECC140*.

Even and in the range 10 through *kMaxDataMatrixModelSize* if the version specified is *eECC200*.

For AIM-compliant rectangular symbols (version ECC 200 only), *ncols* must be one of the following combinations:

nrows	ncols
8	18
8	32
12	26
12	36
16	36
16	48

ecc The ECC level.

symbolPolarity The polarity of the symbol, either dark-on-light or light-on-dark.

angle The angle of the symbol relative to the model. Should be specified in radians.

scale The scale of the symbol relative to the model. *scale* should be greater than 0.

aimFlag If true, then the symbol is AIM-compliant.

mirror If true, then the tool reverses the image of the symbol after acquisition.

Throws

ccAcuSymbolDefs::BadParams
The conditions described above are not met.

Operators

operator==

```
bool operator== (
   const ccAcuSymbolDataMatrixLearnParams &that) const;
```

Compares this object with another. Two **ccAcuSymbolDataMatrixLearnParams** objects are considered equal if all their corresponding data members are identical.

Public Member Functions

aimFlag

```
bool aimFlag() const;
```

Gets the AIM-compliance flag.

cols

```
c_UInt8 cols() const;
```

Gets the number of columns in the symbol grid.

ecc

```
ccAcuSymbolDataMatrixDefs::ECCType ecc()
   const;
```

Gets the Data Matrix version specified by ECC level.

modelTypeAndSize

```
void modelTypeAndSize (
   ccAcuSymbolDataMatrixDefs::ECCType ecc,
   c_UInt8 rows,
   c_UInt8 cols
   bool aimFlag);
```

Sets the version, number of rows, and number of columns in the symbol grid; it also specifies AIM compliance.

Parameters

ecc — The Data Matrix version specified by ECC level.

nrows — The number of rows in the symbol. For AIM-compliant symbols, the number of rows must be the same as the number of columns for the symbol grid for all versions except *eECC200*.

For AIM-compliant square symbols, *nrows* should be:

Odd and in the range 9 through *kMaxDataMatrixModelSize* if the ECC type specified is *eECC0-eECC140*.

Even and in the range 10 through *kMaxDataMatrixModelSize* if the version specified is *eECC200*.

For AIM-compliant rectangular symbols (version ECC 200 only), *nrows* must be one of the following combinations:

nrows	ncols
8	18
8	32
12	26
12	36
16	36
16	48

rows and *cols* should be in the range 9 through *kMaxDataMatrixModelSize*, but need not be equal or have a pre-determined size.

ncols

The number of columns in the symbol. For AIM-compliant symbols, the number of rows must be the same as the number of columns for the symbol grid for all versions except *eECC200*.

For AIM-compliant square symbols, *ncols* should be:

Odd and in the range 9 through *kMaxDataMatrixModelSize* if the ECC type specified is *eECC0-eECC140*.

Even and in the range 10 through *kMaxDataMatrixModelSize* if the version specified is *eECC200*.

For AIM-compliant rectangular symbols (version ECC 200 only), *ncols* must be one of the following combinations:

nrows	ncols
8	18
8	32
12	26
12	36
16	36
16	48

rows and *cols* should be in the range 9 through *kMaxDataMatrixModelSize*, but need not be equal or have a pre-determined size.

aimFlag The AIM-compliance flag. It is learned only when the tool operates in high-contrast mode. Otherwise, you must supply a flag value. For AIM-compliant square symbols, *aimFlag* must be set true. For non-AIM-compliant symbols, *aimFlag* must be set false.

Throws

ccAcuSymbolDefs::BadParams
The conditions described above are not met.

Notes

For all versions except ECC 200, the number of rows must be the same as the number of columns for the symbol grid.

The row and column specification does not include the quiet zone.

rows

```
c_UInt8 rows() const;
```

Gets the number of rows in the symbol grid.

symbolPolarity

```
ccAcuSymbolDataMatrixDefs::Polarity symbolPolarity()
   const;

void symbolPolarity (
   ccAcuSymbolDataMatrixDefs::Polarity pol);
```

- ```
  ccAcuSymbolDataMatrixDefs::Polarity symbolPolarity()
     const;
  ```

  Gets the symbol polarity.

- ```
  void symbolPolarity (
     ccAcuSymbolDataMatrixDefs::Polarity pol);
  ```

  Sets the symbol polarity.

  Parameters

  pol The polarity of the symbol, either dark on light or light on dark.

ccAcuSymbolDataMatrixLearnParams

ccAcuSymbolDataMatrixTool

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolDataMatrixTool: public ccAcuSymbolTool;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

You use this class to create the central object for the Data Matrix tool, which contains the trained representation of a Data Matrix symbol, and the functions for the learn and decode phases of the tool's operation.

Constructors/Destructors

ccAcuSymbolDataMatrixTool

```
ccAcuSymbolDataMatrixTool(
  ccAcuSymbolDefs::OperatingMode opMode =
  ccAcuSymbolDefs::eAutoDetect);

virtual ~ccAcuSymbolDataMatrixTool();

ccAcuSymbolDataMatrixTool(
  const ccAcuSymbolDataMatrixLearnParams& learnParams,
  c_UInt32 learnFlags,
  ccAcuSymbolDefs::OperatingMode opMode =
  ccAcuSymbolDefs::eAutoDetect );
```

- `ccAcuSymbolDataMatrixTool(`
  `ccAcuSymbolDefs::OperatingMode opMode =`
  `ccAcuSymbolDefs::eAutoDetect);`

  Constructs a Symbol tool for decoding Data Matrix symbols. The tool is in an unlearned state, the operating mode defaults to *eAutoDetect*, and all members are initialized to the following defaults:

Parameter	Value
initialLearnParams	default constructed **ccAcuSymbolDataMatrixLearnParams**
bestLearnedParams	default constructed **ccAcuSymbolDataMatrixLearnParams**
learnFlags	*ccAcuSymbolDataMatrixDefs::eAll*
isLearned	false

- `virtual ~ccAcuSymbolDataMatrixTool();`

  Destroys instances of this class.

- ```
  ccAcuSymbolDataMatrixTool(
    const ccAcuSymbolDataMatrixLearnParams& learnParams,
    c_UInt32 learnFlags,
    ccAcuSymbolDefs::OperatingMode opMode =
    ccAcuSymbolDefs::eAutoDetect);
  ```

  Creates a Symbol tool that learns those parameters in the **ccAcuSymbolDataMatrixLearnParams** object specified by *learnFlags* and sets the remaining parameters to their defaults or to those values supplied by the setters of the **ccAcuSymbolDataMatrixLearnParams** class. (Constructor overload)

  Requires that **learnFlags** be a bitwise combination of any of the **ccAcuSymbolDataMatrixDefs::LearnFlags** values. *learnFlags* must be in the range [0, 31].

  Parameters

  learnParams — Parameters that the tool can learn or that you can set. Examples include ECC level, grid size, nominal grid, and polarity. You must set the Learn parameter *mirrorFlag* or use the default.

  learnFlags — Flags that when set direct the tool to learn a specific Learn parameter.

  opMode — Determines the operating mode for the tool, which can be one of:
  ccAcuSymbolDefs::eStandard
  ccAcuSymbolDefs::eHighContrast
  ccAcuSymbolDefs::eAutoDetect

  Throws

  ccAcuSymbolDefs::BadParams
  learnFlags is not in the range 0 through 31.

Public Member Functions

bestLearnedParams

```
const ccAcuSymbolDataMatrixLearnParams&
  bestLearnedParams() const;
```

Returns the final learned parameters after the tool, which is in Learn and Decode mode, has performed the following actions:

1. Found a symbol in the image
2. Learned the parameters specified by **learnFlags**
3. Created a model from the parameters specified by **learnFlags** and those specified by the **ccAcuSymbolDataMatrixLearnParams** object
```

4. Decoded the symbol

Notes

This function should be used to get the learned parameters after the tool has undergone learning successfully as indicated by **isLearned()** returning true.

Throws

ccAcuSymbolDefs::NotLearned

This function is invoked when the tool has not undergone the learning phase successfully and **isLearned()** returns false.

decode

```
bool decode (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccAcuSymbolFinderParams& findParams,
  ccAcuSymbolResult& result) const;
```

Finds the symbol in the given image and decodes it using the specified Find parameters and internally stored Learn parameters. The result of decoding is returned via the result object. Returns true if successful, false if the operation failed. Requires that the image be bound.

This function places the tool in Standard Decode mode for Data Matrix symbology.

Parameters

image The acquired image. The pel buffer must be bound.

findParams The Find parameters, which the tool uses to locate and orient each runtime symbol. Examples include angle, scale, confusion threshold, and acceptance threshold.

result The result object that contains data derived from the decoding operation including angle, score, and scale.

Notes

In high-contrast mode, the tool ignores the *findParams* value, and instead uses 5% as a scale range and 180 degrees (or **ckPI**/2) as an angle range.

This function supports the use of a **ccTimeout**-based timeout.

Throws

ccAcuSymbolDefs::BadImage

The image is not bound.

ccSecurityViolation

Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

initialLearnParams

```
const ccAcuSymbolDataMatrixLearnParams&
  initialLearnParams() const;

void initialLearnParams(
  const ccAcuSymbolDataMatrixLearnParams& learnParams);
```

- ```
  const ccAcuSymbolDataMatrixLearnParams&
    initialLearnParams() const;
  ```

  Gets the initial parameters used for learning prior to estimation by the learn and decode phase.

- ```
  void initialLearnParams(
    const ccAcuSymbolDataMatrixLearnParams& learnParams);
  ```

  Sets the initial parameters used for learning, prior to estimation by the learn and decode phase.

  Parameters

  learnParams — Parameters that the tool can learn or that you can set. Examples include ECC level, grid size, nominal grid, and polarity. You must set the Learn parameter **mirrorFlag** or use the default.

  Notes

  The setter function should be used to initialize the learn parameters in preparation for the learning phase.

  When operating in high-contrast mode, the tool ignores all initial learn parameters except the cell size range.

learn

```
void learn (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccAcuSymbolFinderParams& findParams,
  ccAcuSymbolResult& result);
```

Estimates the optimal configuration of selected symbol learning parameters as specified by *initialLearnParams* and *learnFlags*. If learning is successful, the tool updates *bestLearnedParams* and decodes the symbol using the internally learned parameters and the *findParams* object. Finally, the tool returns decoding results via the result object. Requires that the image be bound.

This function places the tool in Learn and Decode mode for Data Matrix symbology.

Parameters

image — The acquired image. The pel buffer must be bound.

findParams The Find parameters, which the tool uses to locate and orient each runtime symbol. Examples include angle, scale, confusion threshold, and acceptance threshold.

result The result object that contains data derived from the decoding operation including angle, score, and scale.

Notes

In high-contrast mode, the tool ignores the *findParams* value, and instead uses 5% as a scale range and 180 degrees (or **ckPI**/2) as an angle range.

This function supports the use of a **ccTimeout**-based timeout.

Throws

ccAcuSymbolDefs::BadImage
The image is not bound.

ccSecurityViolation
Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

learnFlags

```
c_UInt32 learnFlags() const;

void learnFlags(c_UInt32 flags);
```

- `c_UInt32 learnFlags() const;`

Gets the learnable parameter flags. The parameter *flags* must be a bitwise combination of any of the *ccAcuSymbolDataMatrixDefs::LearnFlags* values. *flags* must be in the range 0 through 31.

- `void learnFlags(c_UInt32 flags);`

  Sets the learnable parameter flags, which are:

Value	Meaning
eNone = 0	Tool learns no parameters.
eECC = 1	Tool learns ECC level.
eGrid = 2	Tool learns Grid size.
eNominalGrid = 4	Tool learns nominal grid.
ePolarity = 8	Tool learns polarity.
eModel = 16	Turns optimization on.
eAll = 31	Tool learns all learning parameters.
kDefaultLearn = eAll	Default is for the tool to learn all learning parameters.

  The parameter *flags* must be a bitwise combination of any of the *ccAcuSymbolQRCodeDefs::LearnFlags* values. *flags* must be in the range 0 through 31.

  Parameters

  flags — The *learnFlags* that if set, direct the tool to learn a specific Learn parameter.

  Notes

  In high-contrast mode, the tool ignores the *flags* value, unless it is *ccAcuSymbolDataMatrixDefs::eNone,* and uses *ccAcuSymbolDataMatrixDefs::eAll*.

  Throws

  ccAcuSymbolDefs::BadParams
  flags is not in the range 0 through 31.

tune

```
virtual void tune (ccGreyAcqFifo &fifo,
   const cc2Xform &xform,
   const ccAffineSamplingParams &affParams,
```

```
   const ccAcuSymbolFinderParams &findParams,
   const ccAcuSymbolTuneParams &tuneParams,
   ccAcuSymbolTuneResult &tuneResult);

virtual void tune (
   const ccAcuSymbolFinderParams &findParams,
   const ccAcuSymbolTuneParams &startTuneParams,
   ccAcuSymbolTuneResult &tuneResult);
```

- ```
  virtual void tune (ccGreyAcqFifo &fifo,
     const cc2Xform &xform,
     const ccAffineSamplingParams &affParams,
     const ccAcuSymbolFinderParams &findParams,
     const ccAcuSymbolTuneParams &tuneParams,
     ccAcuSymbolTuneResult &tuneResult);
  ```

  This function automatically determines the optimal lighting properties using the supplied **ccGreyAcqFifo** to acquire images. The supplied **cc2Xform** and **ccAffineSamplingParams** are used to extract the region of interest from each acquired image. The supplied **ccAcuSymbolTuneParams** object is used to set the tuning parameters and the supplied **ccAcuSymbolFinderParams** object is used to locate the symbols.

  If the tool is currently in the learned state (**isLearned()** returns true), this function uses the best learned parameters and calls **decode()**; otherwise, this function invokes the **learn()** function.

  The tuned parameters (and the decode results produced using those parameters) are returned through the supplied **ccAcuSymbolTuneResult** object. If tuning is successful, the lighting properties of the supplied **ccGreyAcqFifo** are set to the tuned values. If tuning fails, the original values are restored.

  Parameters

fifo	The fifo to acquire images with.
xform	A **cc2xform** that describes the region of interest.
affParams	The sampling parameters to use to construct the region of interest image.
findParams	The search parameters to use to find the symbol.
tuneParams	The tuning parameters to use for tuning.
tuneResult	A reference to a **ccAcuSymbolTuneResult** object into which the results of tuning are placed.
  ```

Notes

You can override the **update()** function to control the tuning process.

This function supports the use of a **ccTimeout**-based timeout.

Throws

ccSecurityViolation

Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

- ```
  virtual void tune (
     const ccAcuSymbolFinderParams &findParams,
     const ccAcuSymbolTuneParams &startTuneParams,
     ccAcuSymbolTuneResult &tuneResult);
  ```

This function automatically determines the optimal lighting properties using images acquired by calling your override of the **ccAcuSymbolTool::acquireImage()** function. The supplied **ccAcuSymbolTuneParams** object is used to set the tuning parameters and the supplied **ccAcuSymbolFinderParams** object is used to locate the symbols.

The tuned parameters (and the decode results produced using those parameters) are returned through the supplied **ccAcuSymbolTuneResult** object.

Parameters

findParams The search parameters to use to find the symbol.

tuneParams The initial tuning parameters to use for tuning (supplied to your callback function).

tuneResult A reference to a **ccAcuSymbolTuneResult** object into which the results of tuning are placed.

Notes

This function supports the use of a **ccTimeout**-based timeout.

Throws

ccSecurityViolation

Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.
  ```

ccAcuSymbolDataMatrixTool

ccAcuSymbolDefs

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolDefs;
```

A name space that holds enumerations and constants used with the Symbol tool classes.

Enumerations

TuneMethod

```
enum TuneMethod;
```

An enumeration that describes which method (learn or decode) was used during tuning.

Value	Meaning
eLearn	The learn method was used during tuning.
eDecode	The decode method was used during tuning.

TuneExitCode

```
enum TuneExitCode;
```

An enumeration that indicates why tuning was stopped.

Value	Meaning
eNormal	Tuning completed normally.
eTime	Tuning timed out.
eNumSuccess	Tuning was stopped because the specified number of successful decodes without score improvement was reached.

OperatingMode `enum OperatingMode;`

An enumeration that describes operating modes for the Symbol tool.

Value	Meaning
eHighContrast	An operating mode suitable for label, PCB, and high-contrast, direct-mark applications where there is high-contrast within the symbol region and possibly high clutter/confusion outside the symbol region.
eStandard	An operating mode intended for wafer and degraded direct-mark applications where there is the possibility of low contrast or degradation within the symbol region and low clutter/confusion outside the symbol region.
eAutoDetect	The Symbol tool automatically selects and runs in an appropriate operating mode based on your system's security. Use *eAutoDetect* for general, direct-mark applications.

System behavior is dependent on the level of hardware security present in your machine. Two security levels are available: Standard security uses the *2DSymbol* security bit, and High Contrast security uses the *2DSymbolHighContrast* security bit. If you wish to run in *eStandard* mode, you require Standard security in your machine. If you wish to run in *eHighContrast* mode, you require High Contrast security in your machine. *eAutoDetect* mode checks your system security and runs in the appropriate mode.

The following table summarizes the operating modes, hardware security, and system behavior.

Run Mode	Security	System behavior
eStandard	Standard	Runs in *eStandard* mode
	High Contrast	Throws an exception
	Standard and High Contrast	Runs in *eStandard* mode
eHighContrast	Standard	Throws an exception
	High Contrast	Runs in *eHighContrast* mode
	Standard and High Contrast	Runs in *eHighContrast* mode

Run Mode	Security	System behavior
eAutoDetect	Standard	Runs in *eStandard* mode
	High Contrast	Runs in *eHighContrast* mode
	Standard and High Contrast	Runs in combined mode

ccAcuSymbolDefs

ccAcuSymbolFinderParams

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolFinderParams : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class holds a set of parameters used to locate and decode both Data Matrix and QR Code symbols. The parameters are defaulted at construction, and may be read using getters and written using setters.

Constructors/Destructors

ccAcuSymbolFinderParams

```
ccAcuSymbolFinderParams();
```

Creates a **ccAcuSymbolFinderParams** object and sets all parameters to the following default values:.

Parameter	Value
accept	0.5
confusion	0.7
contrast	0.5
angleRange	ckPI/90.0
scaleRange	0.0
aspectDistortion	1.0
angleRef	cc2Vect(1.0, 0.0)

Operators

operator==

```
bool operator== (const ccAcuSymbolFinderParams &that)
const;
```

Returns true if these finder parameters are the same as another set. Two parameter sets are considered equal if all of their attributes are the same.

Public Member Functions

accept

```
double accept() const;
```

Returns the acceptance threshold.

acceptAndConfusion

```
void acceptAndConfusion(double accept, double confusion);
```

Sets the acceptance and confusion thresholds during the search phase, the period when the tool searches for a symbol. Both *accept* and *confusion* must be between 0.0 and 1.0 inclusive, and *accept* must be less than or equal to *confusion*.

Parameters

accept — The acceptance threshold, a score below which the tool rejects a candidate symbol as a symbol.

confusion — The confusion threshold, a score above which the tool accepts the candidate symbol as a symbol.

Notes

Features which return a score below the accept threshold are not considered during the search phase.

The **confusion** parameter specifies the maximum score that an erroneous feature can have and not be considered a symbol. Features that return scores greater than or equal to the confusion threshold are considered to be valid instances of the symbol.

Features that return scores below the confusion threshold but greater than or equal to the accept threshold may or may not be valid instances of the symbol.

Throws

ccAcuSymbolDefs::BadParams
The above conditions are not met.

angleRange

```
ccRadian angleRange() const;
void angleRange(ccRadian range);
```

- `ccRadian angleRange() const;`

  Gets the angle range specified in radians. The angle range is a range of angles through which the Symbol tool rotates the symbol model during the search phase. If the angle of the runtime symbol matches the angle of the symbol model after being rotated, the Symbol tool considers the runtime symbol to be an instance of the symbol if all other Find parameters have acceptable values. Requires that the range must be greater than or equal to 0.0.

- `void angleRange(ccRadian range);`

  Sets the angle range, specified in radians. Requires that the range must be greater than or equal to 0.0.

  Parameters

  range The allowed deviation in radians of the **angle** parameter from its expected value

  Throws

  ccAcuSymbolDefs::BadParams
  The range is not greater than or equal to 0.0.

  Notes

  This parameter sets the range of angles to be considered while locating the symbol. The range of angles considered is:

  - *nominal angle - range*
  - *nominal angle + range*

angleRef

```
cc2Vect angleRef () const;
void angleRef (const cc2Vect &ref);
```

- `cc2Vect angleRef () const;`

  Gets the angle that specifies the reference coordinate system of the symbol. This coordinate system is relative to the x-axis of the current coordinate system of the tool. Requires both **angleRef.x()** and **angleRef.y()** must be in the range -1.0 through 1.0.

- `void angleRef (const cc2Vect &ref);`

  Sets the angle that specifies the reference coordinate system of the symbol. This coordinate system is relative to the x-axis of the current coordinate system of the tool. Requires both **angleRef.x()** and **angleRef.y()** must be in the range -1.0 through1.0.

  Parameters

  ref A vector that specifies the reference angle with respect to the x-axis of the current coordinate system.

  Notes

  You use the **angleRef** parameter when a symbology is aligned with a specific reference orientation. For such cases, you can specify the x and y components of the **angleRef** vector in the reference direction with respect to the x-axis of the current coordinate system. All angles are then be measured with respect to the specified reference orientation.

  Using the **angleRef** parameter assumes that the current coordinate system is orthogonal.

  Throws

  ccAcuSymbolDefs::BadParams
  The above condition is not met.

aspectDistortion

```
double aspectDistortion() const;

void aspectDistortion(double aspectDistortion);
```

- `double aspectDistortion() const;`

  Gets the expected aspect distortion while determining the location of the symbol, with respect to the client coordinate space of the image presented for learn/decode. Requires that the **aspectDistortion** be greater than 0.0.

- `void aspectDistortion(double aspectDistortion);`

  Sets the expected aspect distortion while determining the location of the symbol, with respect to the client coordinate space of the image presented for learn/decode. Requires that the **aspectDistortion** be greater than 0.0.

  Parameters

  aspectDistortion The expected ratio of pixel width to pixel height.

Notes

You should use the default value except with unusual cameras or single-field acquires.

Throws

ccAcuSymbolDefs::BadParams
The above condition is not met.

confusion

```
double confusion() const;
```

Returns the confusion threshold.

contrast

```
double contrast() const;
void contrast(double contrast);
```

- `double contrast() const;`

  Gets minimum acceptable model contrast expressed as a fraction of the trained symbol's contrast.

- `void contrast(double contrast);`

  Sets minimum acceptable model contrast expressed as a fraction of the trained symbol's contrast. Requires the specified value must be between 0.0 and 1.0 inclusive.

  Parameters

  contrast The minimum acceptable image contrast where contrast is the difference in pixel value between the symbol and the background

  Throws

  ccAcuSymbolDefs::BadParams
  The above condition is not met.

scaleRange

```
double scaleRange() const;
void scaleRange(double range);
```

- `double scaleRange() const;`

  Gets the scale range, which is a range of percentages by which the Symbol tool multiplies the size of the model during the search phase. If size of the runtime symbol matches the size of the symbol model after being multiplied by any of these

percentages, the Symbol tool considers the runtime symbol to be an instance of the symbol if all other Find parameters have acceptable values. Requires that the scale must be greater than or equal to 0.0.

- `void scaleRange(double range);`

  Sets the scale range. Requires that the scale must be greater than or equal to 0.0.

  Parameters

  range — The range of variation in the scale parameter while locating the symbol expressed as a percentage. To specify a value of 5%, supply a value of 0.05.

  Throws

  ccAcuSymbolDefs::BadParams — The above condition is not met.

ccAcuSymbolLearnParams

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolLearnParams : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

The **ccAcuSymbolLearnParams** class is an abstract base class from which the tool specific classes, **ccAcuSymbolDataMatrixLearnParams** and **ccAcuSymbolQRCodeLearnParams**, are derived. These classes provide symbology specific parameters for learning.

Constructors/Destructors

The constructors and destructors are used by the derived classes. You should not need to use them in your own applications.

Public Member Functions

angle

```
ccRadian angle() const;

void angle(ccRadian ang);
```

- `ccRadian angle() const;`

  Gets the expected orientation of the symbol to be located.

- `void angle(ccRadian ang);`

  Sets the expected orientation of the symbol to be located. (The expected angle is relative to the client coordinate system of the image.)

  Parameters

  ang The expected angle of the symbol to be located.

cellSizeRange

```
const ccRange &cellSizeRange() const;

void cellSizeRange(const ccRange &range);
```

- `const ccRange &cellSizeRange() const;`

  Returns the range of the cell size for the 2-D symbol. The range is measured in the client units of the learn-time image.

- `void cellSizeRange(const ccRange &range);`

  Sets the range of the cell size for the 2-D symbol. The range is measured in the client units of the learn-time image.

Parameters

range The cell size range.

Throws

ccAcuSymbolDefs::BadParams
The start or end of the cell size range is less than 0.

Notes

The cell size range is not learned. It applies to high-contrast and standard operating modes.

If you supply an empty range (**ccRange::EmptyRange()**, the default value) the tool determines an optimal cell size range. The optimal cell size range varies with operating mode and symbology. The following table shows these default ranges for standard and high-contrast operating modes.

Mode	Symbol type	Range (min)	Range (max)
eStandard	QR Code symbols	7 pixels	14 pixels
	Data Matrix symbols	4.5 pixels	9 pixels
eHighContrast	Continuous symbols	7 pixels	21 pixels
	Dot-matrix symbols	7 pixels	14 pixels

Setting a cell size range larger than these default ranges can substantially increase the tool execution time during learning operations.

When loading from older archives, the cell size range is set to its default value.

mirrorFlag

```
bool mirrorFlag() const;
void mirrorFlag(bool mirror);
```

- `bool mirrorFlag() const;`

  Gets the mirror flag.

- `void mirrorFlag(bool mirror);`

  Sets the mirror flag. If **mirror** is true, then mirroring is present, and the tool automatically reverses the image after acquisition.

  Parameters

  mirror — Indicates whether the images containing the symbols to be decoded are mirrored or not.

  Notes

  The *mirror* parameter is learned only when the tool operates in high-contrast mode. Otherwise, a user input is required.

nominalGrid

```
const ccAffineRectangle& nominalGrid () const;
void nominalGrid (const ccAffineRectangle& affRect);
```

- `const ccAffineRectangle& nominalGrid () const;`

  Gets the affine rectangle which defines the 2D symbol grid.

- `void nominalGrid (const ccAffineRectangle& affRect);`

  Sets the affine rectangle which defines the 2D symbol grid.

  Parameters

  affRect — The affine rectangle that defines the 2D symbol grid.

  Notes

  The affine rectangle specified should be defined to enable the tool to determine the dimensions and orientation of the finder pattern and the 2D symbol grid of the symbol to be located. The specification of an affine rectangle allows variations in the scale, orientation, and aspect from the symbol model used to locate the symbol

in the runtime image. These variations typically arise from two sources: (1) From the characteristics of the camera used to acquire the image of the printed symbol, and (2) during printing of the symbol.

The nominal grid of a Data Matrix symbol is defined by the 3 corners of its L-shaped finder pattern. The nominal grid of a Model 1 or Model 2 QR Code symbol is defined by the centers of the 3 finder patterns, whereas the nominal grid of a Micro QR Code symbol is defined by the corners formed by the single finder pattern and the 2 timing patterns running along the top row and left column of the symbol.

scale

```
double scale() const;
void scale(double scale);
```

- `double scale() const;`

Gets the ratio of the expected scale of the symbol to be located in relation to the scale of the model. Requires *scale* must be greater than 0.0.

- `void scale(double scale);`

Sets the ratio of the expected scale of the symbol to be located in relation to the scale of the model. Requires *scale* must be greater than 0.0. (The scale of the symbol used in the learning phase becomes the scale of the model.)

Parameters

scale — The expected scale of the symbol to be located in relation to the scale of the model.

Throws

ccAcuSymbolDefs::BadParams — The scale is not greater than 0.0.

Notes

This parameter is relative to the client coordinate space of the image presented for learn/decode. For example, specifying scale to be 1.2 implies that the expected scale of the symbol (in client coordinate space) is 1.2 times the scale of the model (in client coordinate space).

Operators

operator==

```
bool operator== (const ccAcuSymbolLearnParams &that) const;
```

Returns true if these learn parameters are the same as another set. Two parameter sets are considered equal if all of their attributes are the same.

ccAcuSymbolLearnParams

ccAcuSymbolQRCodeDefs

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolQRCodeDefs: public ccAcuSymbolDefs;
```

A name space that holds enumerations and constants used with the Symbol tool classes.

Enumerations

Polarity

```
enum Polarity;
```

Values for the Polarity parameter:

Value	Meaning
eDarkOnLight	Dark symbol on a light background
eLightOnDark	Light symbol on a dark background

QRModelType

```
enum QRModelType;
```

Values for the QRModelType parameter:

Value	Meaning
eQRModelUnknown = 0	Unknown model
eQRModel1 = 1	*Model 1*
eQRModel2 = 2	*Model 2*
eMicroQR = 4	*Micro QR code*
kDefaultQRModel = eQRModel2	*Model 2*

LearnFlags

```
enum LearnFlags;
```

Setting the following LearnFlags determines which parameters the tool learns:

Value	Meaning
eNone = 0	Tool learns no parameters.
eQRModel = 1	Tool learns QR model.

Value	Meaning
eGrid = 2	Tool learns grid size.
eNominalGrid = 4	Tool learns nominal grid.
ePolarity = 8	Tool learns polarity.
eModel = 16	Turns optimization on.
eAll = 31	Tool learns all learning parameters.
kDefaultLearn = eAll	Default is for the tool to learn all parameters.

kMaxQRModelSize

```
enum kMaxQRModelSize;
```

Maximum size for a QR Code symbol:

Value	Meaning
kMaxQRModelSize = 49	The maximum supported symbol size is 49.

ccAcuSymbolQRCodeLearnParams

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolQRCodeLearnParams:
  public ccAcuSymbolLearnParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The class **ccAcuSymbolQRCodeLearnParams** supplies the parameters that the Symbol tool uses to create a QR Code symbol model. The tool can acquire values for the parameters in this class in the following ways:

- You can set the Learn parameters to their default values by using the default constructor.
- You can specify values for specific Learn parameters using the overload constructor.
- You can set values for the Learn parameters using the setters of the class.
- You can have the tool learn the parameters specified by *learnFlags*. For those parameters not learned by the tool, you use this class to specify the values.

Constructors/Destructors

ccAcuSymbolQRCodeLearnParams

```
ccAcuSymbolQRCodeLearnParams();

virtual ~ccAcuSymbolQRCodeLearnParams();

ccAcuSymbolQRCodeLearnParams(
   const ccAffineRectangle& nominalGrid,
   c_Int32 size,
   ccAcuSymbolQRCodeDefs::QRModelType model =
      ccAcuSymbolQRCodeDefs::kDefaultQRModel,
   ccAcuSymbolQRCodeDefs::Polarity symbolPolarity =
      ccAcuSymbolQRCodeDefs::eDarkOnLight,
```

```
   ccRadian angle = ccRadian(0.0),
   double scale = 1.0,
   bool mirror = false);
```

- `ccAcuSymbolQRCodeLearnParams();`

  Default constructor. The members are initialized to their default values, which are:

Parameter	Value
size	21
nominalGrid	points at (0, 21), (21, 21), (0, 0)
qrModelType	*ccAcuSymbolQRCodeDefs::kDefaultQRModel*
symbolPolarity	*ccAcuSymbolQRCodeDefs::eDarkOnLight*

  Notes

  The nominal grid as well as the number of rows and columns are set to match the smallest symbol size (21 x 21) allowed for a QR Code model 2 symbol. The transform of the nominal grid is set to identity. *qrModelsToLearn* is set to *eQRModel2 | eMicroQR* by default.

- `virtual ~ccAcuSymbolQRCodeLearnParams();`

  Destroys instances of this class.

- ```
  ccAcuSymbolQRCodeLearnParams(
     const ccAffineRectangle& nominalGrid,
     c_Int32 size,
     ccAcuSymbolQRCodeDefs::QRModelType model =
        ccAcuSymbolQRCodeDefs::kDefaultQRModel,
     ccAcuSymbolQRCodeDefs::Polarity symbolPolarity =
        ccAcuSymbolQRCodeDefs::eDarkOnLight,
  ```
  ```

```
ccRadian angle = ccRadian(0.0),
double scale = 1.0,
bool mirror = false);
```

Constructor overload with specified values. *qrModelsToLearn* is set to its default value of *eQRModel2 | eMicroQR*.

The function has the following requirements:

- *model* should be one of the predefined types: *unknown*, *model1, model 2*. or *microQR*
- *size* should be:
  - in the range 21 through *kMaxQRModelSize* for *eQRModel1*
  - in the range 21 through *kMaxQRModelSize* for *eQRModel2*
  - in the range 11 through 17 for *eMicroQR*

Parameters

nominalGrid	The affine rectangle that encloses the symbol.
size	The grid size where the length and width of the symbol are equal.
model	*unknown*, *model1, model 2*. or *microQR*.
symbolPolarity	The polarity of the symbol, either dark on light or light on dark.
angle	The angle of the symbol relative to the model.
scale	The scale of the symbol relative to the model.
mirror	If true, then the tool reverses the image of the symbol after acquisition.

Throws

ccAcuSymbolDefs::BadParams
The conditions described above are not met.

Operators

operator==

```
bool operator== (
   const ccAcuSymbolQRCodeLearnParams &that) const;
```

Compares this object with another. Two **ccAcuSymbolQRCodeLearnParams** objects are considered equal if all their corresponding data members are identical.

Public Member Functions

modelTypeAndSize

```
void modelTypeAndSize (
   ccAcuSymbolQRCodeDefs::QRModelType modelType,
   c_Int32 size);
```

Sets the QR Code model type and grid size. The function has the following requirements:

- *modelType* should be one of the predefined types: *unknown*, *model1, model 2*. or *microQR*
- *size* should be:
  - in the range 21 through *kMaxQRModelSize* for *eQRModel1*
  - in the range 21 through *kMaxQRModelSize* for *eQRModel2*
  - in the range 11 through 17 for *eMicroQR*

Parameters

modelType *unknown*, *model1, model 2*. or *microQR*.

size The grid size where the length and width of the symbol are equal.

Notes

The number of rows is always equal to the number of columns.

Throws

ccAcuSymbolDefs::BadParams
The conditions described above are not met.

qrModel

```
ccAcuSymbolQRCodeDefs::QRModelType qrModel()
   const;
```

Gets the QR Code model type.

size

```
c_Int32 size() const;
```

Gets the grid size.

Notes

The number of rows is always equal to the number of columns.

symbolPolarity

```
ccAcuSymbolQRCodeDefs::Polarity symbolPolarity()
   const;

void symbolPolarity (
   ccAcuSymbolQRCodeDefs::Polarity pol);
```

- ```
  ccAcuSymbolQRCodeDefs::Polarity symbolPolarity()
     const;
  ```

  Gets the symbol polarity, which is either dark-on-light or light-on-dark.

- ```
  void symbolPolarity (
     ccAcuSymbolQRCodeDefs::Polarity pol);
  ```

  Sets the symbol polarity.

  Parameters

  pol The polarity of the symbol, either dark-on-light or light-on-dark.

qrModelsToLearn

```
c_UInt32 qrModelsToLearn() const;

void qrModelsToLearn(c_UInt32 modelsToLearn);
```

Get/Set the QR Code model types to look for at learn time when the model type of the symbol is to be determined by the tool. When the model type need not be learned, this parameter is ignored and *qrModelType* is used instead.

The model types are specified as a bitwise-OR of the appropriate entries from *ccAcuSymbolQRCodeDefs::QRModelType*.

- ```
  c_UInt32 qrModelsToLearn() const;
  ```

  Returns the QR Code model types.

- ```
  void qrModelsToLearn(c_UInt32 modelsToLearn);
  ```

  Sets the QR Code model types.

  Parameters

  modelsToLearn The model types.

Throws

ccAcuSymbolDefs::BadParams

If *modelsToLearn* = 0,
or if *modelsToLearn* is a value not created by the bitwise-OR of values from *ccAcuSymbolQRCodeDefs::QRModelType*.

Notes

This parameter is not learned.

At decode time the tool can only decode a symbol with a model type that it has learned, as specified by the internally stored learned params.

ccAcuSymbolQRCodeTool

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolQRCodeTool : public ccAcuSymbolTool;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

You use this class to create the central object for the QR Code tool, which contains the trained representation of a QR Code symbol, and the functions for the learn and decode phases of the tool's operation.

Constructors/Destructors

ccAcuSymbolQRCodeTool

```
ccAcuSymbolQRCodeTool (
  ccAcuSymbolDefs::OperatingMode opMode =
  ccAcuSymbolDefs::eAutoDetect);

virtual ~ccAcuSymbolQRCodeTool();

ccAcuSymbolQRCodeTool (
  const ccAcuSymbolQRCodeLearnParams& learnParams,
  c_UInt32 learnFlags,
  ccAcuSymbolDefs::OperatingMode opMode =
  ccAcuSymbolDefs::eAutoDetect );
```

- ```
  ccAcuSymbolQRCodeTool (
    ccAcuSymbolDefs::OperatingMode opMode =
    ccAcuSymbolDefs::eAutoDetect);
  ```

  Constructs a Symbol tool for decoding QR Code symbols. The tool is in an unlearned state, the operating mode defaults to *eAutoDetect*, and all members are initialized to the following defaults:

Parameter	Value
initialLearnParams	default constructed **ccAcuSymbolQRCodeLearnParams**
bestLearnedParams	default constructed **ccAcuSymbolQRCodeLearnParams**
learnFlags	*ccAcuSymbolQRCodeDefs::eAll*
isLearned	false

- ```
  virtual ~ccAcuSymbolQRCodeTool();
  ```

  Destroys instances of this class.

- ```
  ccAcuSymbolQRCodeTool (
    const ccAcuSymbolQRCodeLearnParams& learnParams,
    c_UInt32 learnFlags,
    ccAcuSymbolDefs::OperatingMode opMode =
    ccAcuSymbolDefs::eAutoDetect);
  ```

  Creates a Symbol tool that learns those parameters in the **ccAcuSymbolQRCodeLearnParams** object specified by *learnFlags* and sets the remaining parameters to their defaults or to those values supplied by the setters of the **ccAcuSymbolQRCodeLearnParams** class. (Constructor overload)

  Requires that *learnFlags* be a bitwise combination of any of the *ccAcuSymbolQRCodeDefs::LearnFlags* values. *learnFlags* must be in the range 0 through 31.

  Parameters

  learnParams — Parameters that the tool can learn or you can set. Examples include QRModel type, grid size, nominal grid, and polarity. You must set the Learn parameter *mirrorFlag* or use the default.

  learnFlags — Flags that when set direct the tool to learn a specific Learn parameter. Values set in the **ccAcuSymbolQRCodeLearnParams** object are ignored by the **learn()** function unless the appropriated *learnFlags* value is set.

  opMode — Determines the operating mode for the tool, which can be one of:
  ccAcuSymbolDefs::eStandard
  ccAcuSymbolDefs::eHighContrast
  ccAcuSymbolDefs::eAutoDetect

  Throws

  ccAcuSymbolDefs::BadParams
  learnFlags is not in the range 0 through 31.

Public Member Functions

bestLearnedParams

```
const ccAcuSymbolQRCodeLearnParams&
   bestLearnedParams() const;
```

Returns the final learned parameters after the tool, which is in Learn and Decode mode, has performed the following actions:

1. Found a symbol in the image.
2. Learned the parameters specified by *learnFlags*.
```

3. Created a model from the parameters specified by *learnFlags* and those specified by the **ccAcuSymbolQRCodeLearnParams** object.
4. Decoded the symbol.

Notes

This function should be used to get the learned parameters after the tool has undergone learning successfully as indicated by **isLearned()** returning true.

Throws

ccAcuSymbolDefs::NotLearned
This function is invoked when the tool has not undergone the learning phase successfully and **isLearned()** returns false.

decode

```
bool decode (
   const ccPelBuffer_const<c_UInt8>& image,
   const ccAcuSymbolFinderParams& findParams,
   ccAcuSymbolResult& result) const;
```

Finds the symbol in the given image and decodes it using the specified Find parameters and internally stored Learn parameters. The result of decoding is returned via the result object. Returns true if successful; false if the operation failed. Requires that the image be bound.

This function places the tool in Standard Decode mode for QR Code symbology.

Parameters

image	The acquired image. The pel buffer must be bound.
findParams	The Find parameters, which the tool uses to locate and orient each runtime symbol. Examples include angle, scale, confusion threshold, and acceptance threshold.
result	The result object that contains data derived from the decoding operation including angle, score, and scale.

Notes

In high-contrast mode, the tool ignores the *findParams* value, and instead uses 5% as a scale range and 180 degrees (or **ckPI**/2) as an angle range.

This function supports the use of a ccTimeout-based timeout.

Throws

ccAcuSymbolDefs::BadImage
The image is not bound.

ccSecurityViolation
Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

initialLearnParams

```
const ccAcuSymbolQRCodeLearnParams& initialLearnParams()
   const;

void initialLearnParams(
   const ccAcuSymbolQRCodeLearnParams& learnParams);
```

- `const ccAcuSymbolQRCodeLearnParams& initialLearnParams()`
  `const;`

  Gets the initial parameters used for learning prior to estimation by the learn and decode phase.

- `void initialLearnParams(`
  `const ccAcuSymbolQRCodeLearnParams& learnParams);`

  Sets the initial parameters used for learning prior to estimation by the learn and decode phase.

  Parameters

  learnParams Parameters that the tool can learn or that you can set. Examples include QRModel type, grid size, nominal grid, and polarity. You must set the Learn parameter *mirrorFlag* or use the default.

  Notes

  The setter function should be used to initialize the learn parameters in preparation for the learning phase.

  When operating in high-contrast mode, the tool ignores all initial learn parameters except for the cell size range and *qrModelsToLearn*.

learn

```
void learn (const ccPelBuffer_const<c_UInt8>& image,
   const ccAcuSymbolFinderParams& findParams,
   ccAcuSymbolResult& result);
```

Estimates the optimal configuration of selected symbol learning parameters as specified by *initialLearnParams* and *learnFlags*. If learning is successful, the tool updates *bestLearnedParams* and decodes the symbol using the internally learned parameters and the *findParams* object. Finally, the tool returns decoding results via the result object. Requires that the image be bound.

This function places the tool in Learn and Decode mode for QR Code symbology.

Parameters

image The acquired image. The pel buffer must be bound.

findParams The Find parameters, which the tool uses to locate and orient each runtime symbol. Examples include angle, scale, confusion threshold, and acceptance threshold.

result The result object that contains data derived from the decoding operation including angle, score, and scale.

Notes

In high-contrast mode, the tool ignores the *findParams* value, and instead uses 5% as a scale range and 180 degrees (or **ckPI**/2) as an angle range.

This function supports the use of a ccTimeout-based timeout.

Throws

ccAcuSymbolDefs::BadImage
The image is not bound.

ccSecurityViolation
Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

ccAcuSymbolDefs::BadParams
If there are any inconsistencies between the learnable parameter flags and the appropriate learn parameters. For example, the *eQRModel* bit of the learn flags is set to 0 (meaning the model type of the symbol should not be learned) but the model type provided by the initial learn parameters is *eQRModelUnknown*.

learnFlags

```
c_UInt32 learnFlags() const;
void learnFlags(c_UInt32 flags);
```

- `c_UInt32 learnFlags() const;`

  Gets the learnable parameter flags, a bitwise combination of any of the *ccAcuSymbolQRCodeDefs::LearnFlags* values. *learnFlags_* must be in the range 0 through 31.

- `void learnFlags(c_UInt32 flags);`

  Sets the learnable parameter flags, which are:

Value	Meaning
eNone = 0	Tool learns no parameters.
eQRModel = 1	Tool learns QR model.
eGrid = 2	Tool learns grid size.
eNominalGrid = 4	Tool learns nominal grid.
ePolarity = 8	Tool learns polarity.
eModel = 16	Turns optimization on.
eAll = 31	Tool learns all parameters.
kDefaultLearn = eAll	Default is for the tool to learn all parameters.

  The parameter *flags* must be a bitwise combination of any of the *ccAcuSymbolQRCodeDefs::LearnFlags* values. *flags* must be in the range 0 through 31.

  Parameters

  flags — The *learnFlags* that when set direct the tool to learn a specific Learn parameter.

  Notes

  In high-contrast mode, the tool ignores the *flags* value, unless it is *ccAcuSymbolQRCodeDefs::eNone*, and uses *ccAcuSymbolQRCodeDefs::eAll*.

  Throws

  ccAcuSymbolDefs::BadParams — *flags* is not in the range 0 through 31.

tune

```
virtual void tune (ccGreyAcqFifo &fifo,
   const cc2Xform &xform,
   const ccAffineSamplingParams &affParams,
   const ccAcuSymbolFinderParams &findParams,
   const ccAcuSymbolTuneParams &tuneParams,
   ccAcuSymbolTuneResult &tuneResult);

virtual void tune (
   const ccAcuSymbolFinderParams &findParams,
   const ccAcuSymbolTuneParams &startTuneParams,
   ccAcuSymbolTuneResult &tuneResult);
```

- ```
  virtual void tune (ccGreyAcqFifo &fifo,
     const cc2Xform &xform,
     const ccAffineSamplingParams &affParams,
     const ccAcuSymbolFinderParams &findParams,
     const ccAcuSymbolTuneParams &tuneParams,
     ccAcuSymbolTuneResult &tuneResult;
  ```

  This function automatically determines the optimal lighting properties using the supplied **ccGreyAcqFifo** to acquire images. The supplied **cc2Xform** and **ccAffineSamplingParams** are used to extract the region of interest from each acquired image. The supplied **ccAcuSymbolTuneParams** object is used to set the tuning parameters and the supplied **ccAcuSymbolFinderParams** object is used to locate the symbols.

  If the tool is currently in the learned state (**isLearned()** returns true), this function uses the best learned parameters and calls **decode()**; otherwise, this function invokes the **learn()** function.

  The tuned parameters (and the decode results produced using those parameters) are returned through the supplied **ccAcuSymbolTuneResult** object. If tuning is successful, the lighting properties of the supplied **ccGreyAcqFifo** are set to the tuned values. If tuning fails, the original values are restored.

  Parameters

fifo	The fifo to acquire images with.
xform	A **cc2xform** that describes the region of interest.
affParams	The sampling parameters to use to construct the region of interest image.
findParams	The search parameters to use to find the symbol.
tuneParams	The tuning parameters to use for tuning.
tuneResult	A reference to a **ccAcuSymbolTuneResult** object into which the results of tuning are placed.
  ```

Notes

You can override the **update()** function to control the tuning process.

This function supports the use of a ccTimeout-based timeout.

Throws

ccSecurityViolation

Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

ccAcuSymbolDefs::BadParams

if, when **learn()** is invoked, there are any inconsistencies between the learnable parameter flags and the appropriate learn parameters. For example, the *eQRModel* bit of the learn flags is set to 0 (meaning the model type of the symbol should not be learned) but the model type provided by the initial learn parameters is *eQRModelUnknown*.

- ```
  virtual void tune (
     const ccAcuSymbolFinderParams &findParams,
     const ccAcuSymbolTuneParams &startTuneParams,
     ccAcuSymbolTuneResult &tuneResult);
  ```

This function automatically determines the optimal lighting properties using images acquired by calling your override of the **ccAcuSymbolTool::acquireImage()** function. The supplied **ccAcuSymbolTuneParams** object is used to set the tuning parameters and the supplied **ccAcuSymbolFinderParams** object is used to locate the symbols.

The tuned parameters (and the decode results produced using those parameters) are returned through the supplied **ccAcuSymbolTuneResult** object.

Parameters

findParams The search parameters to use to find the symbol.

tuneParams The initial tuning parameters to use for tuning (supplied to your callback function).

tuneResult A reference to a **ccAcuSymbolTuneResult** object into which the results of tuning are placed.

Notes

This function supports the use of a ccTimeout-based timeout.

Throws
  ```

ccSecurityViolation

Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

ccAcuSymbolDefs::BadParams

if, when **learn()** is invoked, there are any inconsistencies between the learnable parameter flags and the appropriate learn parameters. For example, the *eQRModel* bit of the learn flags is set to 0 (meaning the model type of the symbol should not be learned) but the model type provided by the initial learn parameters is *eQRModelUnknown*.

ccAcuSymbolResult

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolResult : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the results for either Data Matrix and QR Code symbols.

Constructors/Destructors

ccAcuSymbolResult

```
ccAcuSymbolResult();
```

Constructs a **ccAcuSymbolResult** object with all members initialized to the following default values:

Parameter	Value
isFound	False
decodedFlag	False
decodedMBCSString	empty vector
decodedString	" "
decodedData	0
location	cc2Vect(0.0, 0.0)
score	0.0
scale	1.0
angle	0.0
aspect	1.0
time	0.0

Parameter	Value
imgFromClient	Identity transform
resultGrid	Default constructed affine rectangle
numErrors	0
numErrorBits	0
learnParams	Default constructed **ccAcuSymbolLearnParams**

Operators

operator==

```
bool operator==(const ccAcuSymbolResult&) const;
```

Returns true if all result parameters except for *time* are the same as another set. Two parameter sets are considered equal if all of their attributes are the same.

Public Member Functions

angle

```
ccRadian angle() const;
```

Returns the angle with respect to the reference angle if any.

aspect

```
double aspect() const;
```

Returns the aspect ratio of the 2D symbol that was found with respect to the client coordinate space. The aspect ratio is the expected ratio of pixel height to pixel width.

decodedData

```
cmStd vector<c_UInt8> decodedData() const;
```

Returns the raw bit-stream to be decoded as a sequence of bytes. You can use this function to return a sequence of uninterpreted bytes, from which the original string encoded in the symbol can be reconstructed using an alternate custom string decoder.

Throws

ccAcuSymbolDefs::BadString
This function is invoked when **isFound()** returns false.

Notes

The raw decoded data corresponds to the encoded bit-stream. It typically contains sub-streams, which begin with a ECI header that contains information about the mode used for encoding and the number of characters in the sub-stream. The

header is followed by the data bit-stream. A raw bit-stream corresponding to a QR Code symbol may be composed of two sub-streams, the first encoded in AlphaNumeric mode, and the second in Kanji (2 bytes per character).

Conceptually, the raw bit-stream could be represented as follows:

```
|AlphaNumeric mode indicator|char count| -- Data bit-stream
--| |Kanji Mode indicator |char count| -- Data bit-stream --|
```

The sequence of bytes in the raw bit-stream should not be altered prior to decoding as this may corrupt any multi-byte encoded data.

decodedString `ccCvlString decodedString() const;`

Returns the decoded string of the symbol just found.

Throws

ccAcuSymbolDefs::BadString
This function is invoked when **isDecoded()** returns false.

Notes

This function processes the raw uninterpreted bit-stream and generates an ANSI string. Use this function only for symbols that you know do not contain multibyte characters.

decodedMBCSString

```
cmStd vector<c_UInt8> decodedMBCSString() const;
```

Returns the decoded data as a null-terminated, multi-byte string. The size of the vector includes the null terminator.

For symbologies that encode 8-bit characters, this function returns a string of single-byte characters. For example, for the Data Matrix symbology, this function returns a single-byte stream of ASCII and extended ASCII characters.

For symbologies that encode single-byte and double-byte characters, this function returns a stream of multi-byte characters. For example, for the QR Code symbology, this function returns a stream of JIS8 characters (8-bit character set) and Shift JIS characters (double-byte character set that encodes Kanji characters).

You can convert multi-byte characters to unicode using the **mbtowc()** and **mbstowcs()** C library routines.

Throws

ccAcuSymbolDefs::BadString
If this function is invoked when **isDecoded()** returns *false*.

imageFromClientXform

```
const cc2Xform& imageFromClientXform () const;
```

Returns the transform required to translate client coordinates to image coordinates.

isDecoded

```
bool isDecoded() const;
```

Returns true if this symbol is decoded, false otherwise.

isFound

```
bool isFound() const;
```

Returns true if this runtime symbol is found, false otherwise. A symbol is found if its score equals or exceeds the accept threshold, implying that the symbol location was determined with sufficient confidence.

location

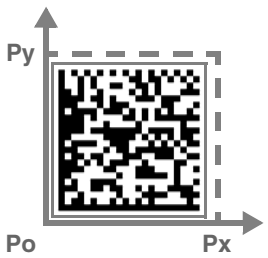
```
cc2Vect location() const;
```

Returns the location of center of the found symbol in the client coordinate system of the runtime image.

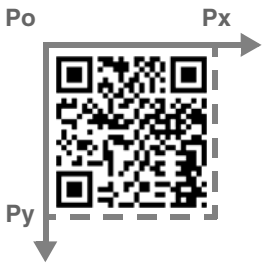
resultGrid

```
const ccAffineRectangle& resultGrid () const;
```

Returns the affine rectangle applied to compute this result. The affine rectangle returned by this function reflects the “natural” orientation and handedness of the symbol being decoded, as shown in the following figure:



Data Matrix (right-handed)



QR Code (left handed)

scale

```
double scale() const;
```

Returns the scale of the 2D symbol that was found, as a fraction of the scale of the model size.

score

```
double score() const;
```

Returns the result score in the range 0.0 through 1.0, which is the correlation between the runtime symbol and the model.

Notes

If the score is 0.0 after the Symbol tool attempts to locate the symbol, the tool does not perform the decoding step. In this case, the **angle**, **scale** and **aspect** are set to their nominal values.

time

```
double time() const;
```

Returns the time (in seconds) required to decode the symbol.

numErrors

```
c_Int32 numErrors () const;
```

Returns the number of erroneous data words encountered while decoding the symbol. The function returns -1 if this information is not available.

numErrorBits

```
c_Int32 numErrorBits () const;
```

Returns the number of erroneous bits encountered while decoding the symbol. This function returns -1 if this information is not available

learnParams

```
const ccAcuSymbolLearnParams& learnParams () const;
```

Returns the **ccAcuSymbolLearnParams** used to compute this result.

Throws

ccAcuSymbolDefs::BadParams

The symbol is neither decoded nor found: both **isFound()** and **isDecoded()** return false.

ccAcuSymbolResult

ccAcuSymbolTool

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolTool : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

ccAcuSymbolTool is an abstract the base class from which the following tool classes representing specific 2D matrix symbologies are derived:

- ccAcuSymbolDataMatrixTool
- ccAcuSymbolQRCodeTool

The Symbol tool operates in two modes:

- Standard Decode mode where the tool uses a set of externally specified parameters to locate and decode the target symbol
- Learn and Decode mode, which consists of a learning phase and a decode phase

  During the learning phase, the application acquires a source image with the target symbol. The tool then locates a candidate symbol in the image and analyzes it, and learns optimal parameters based on learning flags provided. The objective of the learning phase is to estimate a set of parameters when incomplete information regarding best parameters is available.The learned parameters are stored inside the tool along with parameters that you specify as part of a model of the symbol.

  During the decode phase, the tool decodes the symbol using the internally stored parameters. Results of decoding are returned via a **ccAcuSymbolResult** object.

For more detailed information about the Symbol tool, see the chapter *Symbol Tool* in the *CVL Vision Tools Guide*.

Constructors/Destructors

The constructors and destructors are used by the derived classes. You should not need to use them in your own applications.

Public Member Functions

decode

```
virtual bool decode (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccAcuSymbolFinderParams& findParams,
    ccAcuSymbolResult& result) const = 0;
```

Finds the symbol in the given image and decodes it using the specified *findParams* and internally stored learned parameters. The result of decoding is returned via *result*. Returns true if the decoding operation was successful; false if not. Requires that the image be bound.

Notes

The function **decode()** should be over-ridden in the derived tool class. Because this function is pure virtual, you need to define this function in all the classes derived from this class corresponding to the standard decode phase of this tool.

After a symbol is found within a defined area, **Decode()** begins symbol recognition. If the size in found symbols differs significantly (greater than ±5 percent), the function can fail. To prevent this failure, avoid using *eHighContrast* operating mode and adjust the *ScaleRange*, which is one of the *findParams,* in proportion to the difference in size.

Parameters

image	The pel buffer that contains the image being read.
findParams	The finder parameters, which the tool uses to locate and orient each runtime symbol. Examples include angle, scale, confusion threshold, and acceptance threshold.
result	The result object that contains data derived from the decoding operation including angle, score, and scale.

Throws

ccAcuSymbolDefs::BadImage
The image is not bound.

ccSecurityViolation
Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

isLearned

```
bool isLearned() const;
```

Gets the learned state of this tool. True indicates that the tool has undergone the learning phase successfully, while false indicates an unsuccessful learning phase.

learn

```
virtual void learn (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccAcuSymbolFinderParams& findParams,
    ccAcuSymbolResult& result) = 0;
```

Estimates the optimal configuration of selected symbol learning parameters as specified by the **initialLearnParams** and **learnFlags**. The result of decoding is returned via the result object. Requires that the image be bound.

Notes

The function **learn()** should be over-ridden in the derived tool class. Because this function is pure virtual, you need to define this function in all the classes derived from this class corresponding to the standard decode phase of this tool.

Parameters

image The pel buffer that contains the image being read.

findParams The finder parameters, which the tool uses to locate and orient each runtime symbol. Examples include angle, scale, confusion threshold, and acceptance threshold.

result The result object that contains data derived from the decoding operation including angle, score, and scale.

Throws

ccAcuSymbolDefs::BadImage
The image is not bound.

ccSecurityViolation
Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

operatingMode

```
ccAcuSymbolDefs::OperatingMode operatingMode() const;
```

Returns the current operating mode for the tool, which can be one of the following values:

ccAcuSymbolDefs::eStandard
ccAcuSymbolDefs::eHighContrast
ccAcuSymbolDefs::eAutoDetect

---

tune

```
virtual void tune (ccGreyAcqFifo &fifo,
   const cc2Xform &xform,
   const ccAffineSamplingParams &affParams,
```

```
   const ccAcuSymbolFinderParams &findParams,
   const ccAcuSymbolTuneParams &tuneParams,
   ccAcuSymbolTuneResult &tuneResult) = 0;

virtual void tune (
   const ccAcuSymbolFinderParams &findParams,
   const ccAcuSymbolTuneParams &startTuneParams,
   ccAcuSymbolTuneResult &tuneResult) = 0;
```

- ```
  virtual void tune (ccGreyAcqFifo &fifo,
     const cc2Xform &xform,
     const ccAffineSamplingParams &affParams,
     const ccAcuSymbolFinderParams &findParams,
     const ccAcuSymbolTuneParams &tuneParams,
     ccAcuSymbolTuneResult &tuneResult) = 0;
  ```

  This function automatically determines the optimal lighting properties using the supplied **ccGreyAcqFifo** to acquire images. The supplied **cc2Xform** and **ccAffineSamplingParams** are used to extract the region of interest from each acquired image. The supplied **ccAcuSymbolTuneParams** object is used to set the tuning parameters and the supplied **ccAcuSymbolFinderParams** object is used to locate the symbols.

  If the tool is currently in the learned state (**isLearned()** returns true), this function uses the best learned parameters and calls **decode()**; otherwise, this function invokes the **learn()** function.

  The tuned parameters (and the decode results produced using those parameters) are returned through the supplied **ccAcuSymbolTuneResult** object. If tuning is successful, the lighting properties of the supplied **ccGreyAcqFifo** are set to the tuned values. If tuning fails, the original values are restored.

  Parameters

fifo	The fifo to acquire images with.
xform	A **cc2xform** that describes the region of interest.
affParams	The sampling parameters to use to construct the region of interest image.
findParams	The search parameters to use to find the symbol.
tuneParams	The tuning parameters to use for tuning.
tuneResult	A reference to a **ccAcuSymbolTuneResult** object into which the results of tuning are placed.

  Notes

  You can override the **update()** function to control the tuning process.
  ```

Throws

ccSecurityViolation

Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

- ```
  virtual void tune (
     const ccAcuSymbolFinderParams &findParams,
     const ccAcuSymbolTuneParams &startTuneParams,
     ccAcuSymbolTuneResult &tuneResult) = 0;
  ```

  This function automatically determines the optimal lighting properties using images acquired by calling your override of the **acquireImage()** function. The supplied **ccAcuSymbolTuneParams** object is used to set the tuning parameters and the supplied **ccAcuSymbolFinderParams** object is used to locate the symbols.

  The tuned parameters (and the decode results produced using those parameters) are returned through the supplied **ccAcuSymbolTuneResult** object.

  Parameters

  findParams The search parameters to use to find the symbol.

  tuneParams The initial tuning parameters to use for tuning (supplied to your callback function).

  tuneResult A reference to a **ccAcuSymbolTuneResult** object into which the results of tuning are placed.

  Throws

  ccSecurityViolation

  Hardware security is not enabled for the specified operating mode; or hardware security is not enabled for any operating mode.

update

```
virtual bool update (
   const ccAcuSymbolTuneResult& currentTuneResult,
   const ccAcuSymbolTuneResult& bestTuneResult);
```

Provides a hook that lets you update intermediate **tune()** results. The tuning process continues as long as this function returns true. The default **update()** function always returns true.

Parameters

currentTuneResult

The result of the current tune cycle.
```

bestTuneResult The result of the tune cycle that has produced the best (highest scoring) result so far.

userPreProcessAcquiredImage

```
virtual bool userPreprocessAcquiredImage(
   ccPelBuffer<c_UInt8>& src,
   const ccPelRect& rect, ccPelBuffer<c_UInt8>& dst);
```

Provides a hook that lets perform your own preprocessing on images acquired by the tool for reading or tuning. If your override returns false, tuning is stopped.

The default **userPreprocessAcquiredImage()** function copies the pixels in *src* that lie within *rect* to *dst* and returns true.

Parameters

src The acquired image. This pel buffer must be bound.

rect The region of interest within *src*.

dst The destination pel buffer. This pel buffer must be unbound.

acquireImage

```
virtual ccPelBuffer<c_UInt8> acquireImage(
   const ccAcuSymbolTuneParams& tuneParams);
```

You must provide an override of this function if you use the second overload of the **tune()** function. Your implementation should acquire and return an image. The function is supplied with a **ccAcuSymbolTuneParams** you can use to control your acquisition.

Parameters

tuneParams The tuning parameters computed by the **tune()** function.

Throws

ccAcuSymbolDefs::BadImage
You did not provide an override of this function.

ccAcuSymbolTuneParams

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolTuneParams : public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class lets you specify the parameters used to tune the lighting, parameters for decoding symbols etched or scribed on reflective surfaces.

Constructors/Destructors

ccAcuSymbolTuneParams

```
ccAcuSymbolTuneParams (bool enableDark = true,
   double darkLow = 0.062745,
   double darkHigh = 1.0, double darkStep = 0.12157,
   bool enableBright = true, double brightLow = 0.062745,
   double brightHigh = 1.0, double brightStep = 0.12157,
   double nominalLightPower = 0.062745,
   double nominalBrightFieldPowerRatio = 1.0,
   double maxTime = 3000., c_Int32 validLimit = 255);
```

Constructs a **ccAcuSymbolTuneParams**.

Parameters

enableDark	True to enable darkfield tuning.
darkLow	The minimum light power for darkfield tuning. Must be in the range 0.0 through 1.0.
darkHigh	The maximum light power for darkfield tuning. Must be in the range 0.0 through 1.0.
darkStep	The darkfield step value. Must be in the range 0.0 through 1.0.
enableBright	True to enable brightfield tuning.
brightLow	The minimum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

brightHigh The maximum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

brightStep The brightfield step value. Must be in the range 0.0 through 1.0.

nominalLightPower
The nominal light power value (used by the tune callback function).

nominalBrightFieldPowerRatio
The nominal brightfield power ratio (the portion of the light power allocated to brightfield lighting). Must be in the range 0.0 through 1.0. This value is used by the tune callback function.

maxTime The maximum number of seconds to allow for tuning.

validLimit The maximum number of successful decodes that can occur without producing a new best score.

Throws

ccAcuSymbolToolDefs::BadParams
A parameter value was out of range.

Operators

operator==

```
bool operator== (const ccAcuSymbolTuneParams& that) const;
```

Two **ccAcuSymbolTuneParams** objects are considered equal if all of their data members are equal to a numerical precision of 1.e-8.

Parameters

that The **ccAcuSymbolTuneParams** to compare to this one.

Public Member Functions

enableBright

```
bool enableBright() const;
void enableBright(bool val);
```

- `bool enableBright() const;`

  Returns true if brightfield tuning is enabled.

- `void enableBright(bool val);`

  Enables or disables brightfield tuning.

Parameters

val True to enable brightfield tuning, false to disable.

enableDark

```
bool enableDark() const;
void enableDark(bool val);
```

- `bool enableDark() const;`

  Returns true if darkfield tuning is enabled.

- `void enableDark(bool val);`

  Enables or disables darkfield tuning.

  Parameters

  val True to enable darkfield tuning, false to disable.

brightLow

```
double brightLow() const;
void brightLow(double val);
```

- `double brightLow() const;`

  Returns the minimum light power for brightfield tuning.

- `void brightLow(double val);`

  Sets the minimum light power for brightfield tuning.

  Parameters

  val The minimum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuSymbolToolDefs::BadParams
  val was out of range.

ccAcuSymbolTuneParams

brightHigh

```
double brightHigh() const;
void brightHigh(double val);
```

- `double brightHigh() const;`

  Returns the maximum light power for brightfield tuning.

- `void brightHigh(double val);`

  Sets the maximum light power for brightfield tuning.

  Parameters

  val — The maximum light power for brightfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuSymbolToolDefs::BadParams
  val was out of range.

brightStep

```
double brightStep() const;
void brightStep(double val);
```

- `double brightStep() const;`

  Returns the brightfield step for tuning.

- `void brightStep(double val);`

  Sets the brightfield step for tuning.

  Parameters

  val — The brightfield step value. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuSymbolToolDefs::BadParams
  val was out of range.

darkLow

```
double darkLow() const;
void darkLow(double val);
```

- `double darkLow() const;`

  Returns the minimum light power for darkfield tuning.

- `void darkLow(double val);`

  Sets the minimum light power for darkfield tuning.

  Parameters

  val The minimum light power for darkfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuSymbolToolDefs::BadParams
  val was out of range.

darkHigh

```
double darkHigh() const;
void darkHigh(double val);
```

- `double darkHigh() const;`

  Returns the maximum light power for darkfield tuning.

- `void darkHigh(double val);`

  Sets the maximum light power for darkfield tuning.

  Parameters

  val The maximum light power for darkfield tuning. Must be in the range 0.0 through 1.0.

  Throws

  ccAcuSymbolToolDefs::BadParams
  val was out of range.

darkStep

```
double darkStep() const;
void darkStep(double val);
```

- `double darkStep() const;`

  Returns the darkfield step for tuning.t

- `void darkStep(double val);`

  Sets the darkfield step for tuning.

  Parameters
  val The darkfield step value. Must be in the range 0.0 through 1.0.

  Throws
  ccAcuSymbolToolDefs::BadParams
  val was out of range.

nominalLightPower

```
double nominalLightPower() const;
void nominalLightPower(double val);
```

- `double nominalLightPower() const;`

  Returns the nominal light power value.

- `void nominalLightPower(double val);`

  Sets the nominal light power value. This value is used in conjunction with the **ccAcuSymbolTool::tune()** function. It specifies the light power setting for a remote fifo object used to acquire an image during tuning.

  Parameters
  val The light power value in the range 0.0 through 1.0.

  Throws
  ccAcuSymbolToolDefs::BadParams
  val was out of range.

nominalBrightFieldPowerRatio

```
double nominalBrightFieldPowerRatio() const;
void nominalBrightFieldPowerRatio(double val);
```

- `double nominalBrightFieldPowerRatio() const;`

  Returns the nominal brightfield power ratio (the portion, in the range 0.0 through 1.0, of light power allocated to brightfield illumination).

- `void nominalBrightFieldPowerRatio(double val);`

  Sets the nominal brightfield power ratio (the portion, in the range 0.0 through 1.0, of light power allocated to brightfield illumination). This value is used in conjunction with the **ccAcuSymbolTool::tune()** function. It specifies the brightfield power ratio for a remote fifo object used to acquire an image during tuning.

  Parameters

  val The brightfield power ratio in the range 0.0 through 1.0.

  Throws

  ccAcuSymbolToolDefs::BadParams
  val was out of range.

maxTime

```
double maxTime() const;
void maxTime(double maxTime);
```

- `double maxTime() const;`

  Returns the tuning timeout value (in seconds).

- `void maxTime(double maxTime);`

  Sets the tuning timeout value (in seconds). Tuning will be stopped after the specified number of seconds.

  Parameters

  maxTime The timeout value in seconds.

  Throws

  ccAcuSymbolDefs::BadParams if *maxTime* is less than 0.

validLimit

```
c_Int32 validLimit() const;
void validLimit(c_Int32 val);
```

- `c_Int32 validLimit() const;`

  Returns the maximum successful decode count. Tuning is stopped after the indicated number of decodes are performed with no score improvement.

- `void validLimit(c_Int32 val);`

  Sets the maximum successful decode count. Tuning is stopped after the indicated number of decodes are performed with no score improvement.

  Parameters

  val The maximum number of successful decodes in the range 0 through 255.

  Throws

  ccAcuSymbolDefs::BadParams
  val is out of range.

  Notes

  If you specify both a timeout and a valid limit, tuning stops when the first condition is met.

ccAcuSymbolTuneResult

```
#include <ch_cvl/acusymbl.h>

class ccAcuSymbolTuneResult : public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class contains the results of a tune operation performed by the Symbol tool.

Constructors/Destructors

ccAcuSymbolTuneResult

```
ccAcuSymbolTuneResult ();
```

Constructs a **ccAcuSymbolTuneResult** with the following values::

Parameter	Value
isTuned	false
tuneMethod	*ccAcuSymbolDefs::eLearn*
lightPower	0.0
brightFieldPowerRatio	0.12157
result	Default-constructed **ccAcuSymbolResult**
numSuccess	0
time	0
tuneExitCode	*ccAcuSymbolDefs::eNormal*

Operators

operator==

```
bool operator== (const ccAcuSymbolTuneResult& that) const;
```

Two **ccAcuSymbolTuneResult** objects are considered equal if all of their attributes are the same to a numerical precision of 1.e-8.

Parameters

that The **ccAcuSymbolTuneResult** to compare to this one.

Public Member Functions

isTuned

```
bool isTuned() const;
```

Returns true if this **ccAcuSymbolTuneResult** contains the results of a successful tune operation.

tuneMethod

```
ccAcuSymbolDefs::TuneMethod tuneMethod() const;
```

Returns the tuning method used during tuning (learn or decode). This function returns one of the following values:

ccAcuSymbolToolDefs::eLearn
ccAcuSymbolToolDefs::eDecode

lightPower

```
double lightPower() const;
```

Returns the tuned light power level value (in the range 0.0 through 1.0).

brightFieldPowerRatio

```
double brightFieldPowerRatio() const;
```

Returns the tuned bright field power ratio value (in the range 0.0 through 1.0).

result

```
const ccAcuSymbolResult& result() const;
```

Returns the **ccAcuSymbolResult** object containing the decode results produced using the tuned lighting values

numSuccesses

```
c_UInt32 numSuccesses() const;
```

Returns the number of successful learn or decode operations that were performed during tuning without improving the decode score.

time

```
double time() const;
```

Returns the amount of time (in seconds) required for tuning.

exitCode

```
ccAcuSymbolDefs::TuneExitCode exitCode() const;
```

Returns the reason that tuning terminated. This function returns one of the following values:

ccAcuSymbolDefs::eNormal
ccAcuSymbolDefs::eTime
ccAcuSymbolDefs::eNumSuccess

score

```
virtual double score() const;
```

Returns a score value for this **ccAcuSymbolTuneResult** object based on the values of the **ccAcuSymbolResult** object returned by **result()**. You can override this function to provide your own scoring function for use during tuning.

ccAcuSymbolTuneResult

ccAffineRectangle

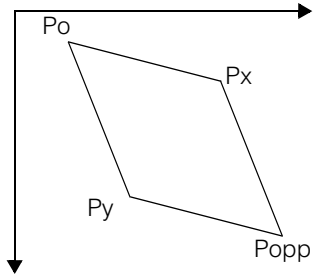
```
#include <ch_cvl/affrect.h>

class ccAffineRectangle : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class describes an affine rectangle (a quadrilateral where the opposite sides are parallel). A **ccAffineRectangle** can be defined by a unit square transformation, by the locations of three of its vertices, or by its location, width, height, and skew angle. Each of the vertices of an affine rectangle are labeled, as shown in the following figure:



Constructors/Destructors

ccAffineRectangle

```
ccAffineRectangle ();

ccAffineRectangle (const cc2Xform& affRectFromUS);

ccAffineRectangle (const cc2Vect& cornerPo,
  const cc2Vect& cornerPx, const cc2Vect& cornerPy);

ccAffineRectangle (const cc2Vect& center, double xLen,
  double yLen, const ccRadian& xRotation = ccRadian(0),
  const ccRadian& skew = ccRadian(0));
```

- `ccAffineRectangle ();`

  Constructs a **ccAffineRectangle** equal to the unit square in client coordinates (a rectangle whose vertices are the points (0,0), (1,0), (1,1), and (0,1)).

- `ccAffineRectangle (const cc2Xform& affRectFromUS);`

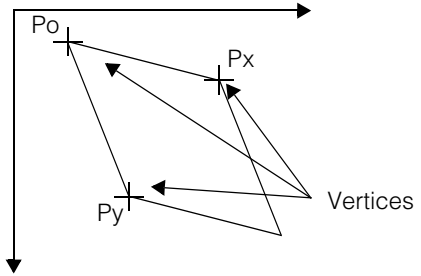
  Constructs the **ccAffineRectangle** defined by the application of the supplied **cc2Xform** to a unit square in client coordinates. A unit square is a rectangle whose vertices are the points (0,0), (1,0), (1,1), and (0,1).

  Parameters

  affRectFromUS A **cc2Xform** that defines this **ccAffineRectangle**.

- `ccAffineRectangle (const cc2Vect& cornerPo, const cc2Vect& cornerPx, const cc2Vect& cornerPy);`

  Constructs the **ccAffineRectangle** defined by the supplied vertices. The following figure shows the locations of these vertices.

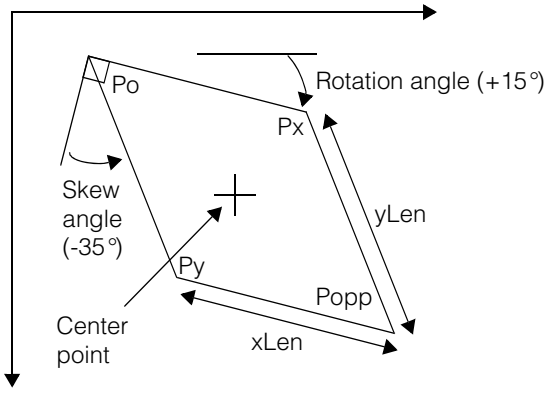


Parameters

cornerPo	The location of Po in client coordinates.
cornerPx	The location of Px in client coordinates.
cornerPy	The location of Py in client coordinates.

- `ccAffineRectangle (const cc2Vect& center, double xLen, double yLen, const ccRadian& xRotation = ccRadian(0), const ccRadian& skew = ccRadian(0));`

Constructs the **ccAffineRectangle** defined by the supplied center point location, width, height, angle of rotation, and angle of skew. The following figure shows how these values define the **ccAffineRectangle**.



Parameters

center	The center of the affine rectangle, in client coordinates
xLen	The length of the Po-Px side of the affine rectangle, in client coordinates
yLen	The length of the Po-Py side of the affine rectangle, in client coordinates
xRotation	The angle between the client coordinate system x-axis and the Po-Px side of the affine rectangle
skew	The angle between a line drawn perpendicular to the Po-Px side of the affine rectangle and the Po-Py side of the affine rectangle

Throws

ccAffineRectangle::BadLength
xLen or *yLen* is less than or equal to 0.

ccAffineRectangle::BadSkew
skew is π/2 radians or -π/2 radians (These two values are illegal; they produce a degenerate affine rectangle.) Note that this error is thrown if *skew* is within **ckAffineRectangleSkewAngleTol** of π/2 radians or -π/2 radians. You can determine if a given skew angle is valid by calling **ccAffineRectangle::isBadSkew()**.

Operators

operator==

```
bool operator== (const ccAffineRectangle& rhs) const;
```

Return true if this **ccAffineRectangle** is the same as the supplied **ccAffineRectangle**.

Two **ccAffineRectangle**s are considered equal if and only if they have the same Po, Px, and Py values.

Parameters
rhs The other **ccAffineRectangle** to compare.

operator!=

```
bool operator!= (const ccAffineRectangle& that) const;
```

Return true if this **ccAffineRectangle** is not the same as the supplied **ccAffineRectangle**.

Two **ccAffineRectangle**s are considered equal if and only if they have the same Po, Px, and Py values.

Parameters
that The other **ccAffineRectangle** to compare.

Public Member Functions

affRectFromUnitSq

```
const cc2Xform& affRectFromUnitSq () const;

void affRectFromUnitSq (const cc2Xform& affRectFromUS);
```

- `const cc2Xform& affRectFromUnitSq () const;`

Returns the **cc2Xform** that defines the transformation from a unit square to the affine rectangle described by this **ccAffineRectangle**.

- `void affRectFromUnitSq (const cc2Xform& affRectFromUS);`

  Sets the **cc2Xform** that defines the transformation from a unit square to the affine rectangle described by this **ccAffineRectangle**.

  Parameters

  affRectFromUS A **cc2Xform** that describes the transformation from the unit square to this **ccAffineRectangle**.

---

cornerPo

```
cc2Vect cornerPo () const;
void cornerPo (const cc2Vect& po);
```

- `cc2Vect cornerPo () const;`

  Returns the location of the Po vertex of this **ccAffineRectangle** in client coordinates.

- `void cornerPo (const cc2Vect& po);`

  Sets the Po vertex of this **ccAffineRectangle**. The remaining dimensions of the **ccAffineRectangle** are unchanged. Use this function to move a **ccAffineRectangle** without changing its size or shape.

  Parameters

  po The location of the Po vertex in client coordinates.

cornerPx

```
cc2Vect cornerPx () const;
```

Returns the location of the Px vertex of this **ccAffineRectangle** in client coordinates.

cornerPy

```
cc2Vect cornerPy () const;
```

Returns the location of the Py vertex of this **ccAffineRectangle** in client coordinates.

cornerPopp

```
cc2Vect cornerPopp () const;
```

Returns the location of the Popp vertex of this **ccAffineRectangle** in client coordinates.

center

```
cc2Vect center () const;
void center (const cc2Vect& ctr);
```

- `cc2Vect center () const;`

  Returns the location of the center of this **ccAffineRectangle** in client coordinates.

- `void center (const cc2Vect& ctr);`

  Sets the center point of this **ccAffineRectangle**. The remaining dimensions of the **ccAffineRectangle** are unchanged. Use this function to move a **ccAffineRectangle** without changing its size or shape.

  Parameters

  ctr The center point of this **ccAffineRectangle** in client coordinates.

xLength

```
double xLength () const;
void xLength (double xLen);
```

- `double xLength () const;`

  Returns the length of the Po-Px side of this **ccAffineRectangle** in client coordinates.

- `void xLength (double xLen);`

  Sets the length of the Po-Px side of this **ccAffineRectangle**. The center point, height, and angles of skew and rotation are unchanged.

  Parameters

  xLen The length of the Po-Px side in client coordinates.

  Throws

  ccAffineRectangle::BadLength
  xLen is less than or equal to 0.

yLength

```
double yLength () const;
void yLength (double yLen);
```

- `double yLength () const;`

  Returns the height of this **ccAffineRectangle** in client coordinates.

- `void yLength (double yLen);`

  Sets the height of this **ccAffineRectangle**. The center point, width, and angles of skew and rotation are unchanged.

  Parameters

  yLen The length of the Po-Py side in client coordinates.

  Throws

  ccAffineRectangle::BadLength
  yLen is less than or equal to 0.

xRotation

```
ccRadian xRotation () const;
void xRotation (const ccRadian& xRot);
```

- `ccRadian xRotation () const;`

  Returns the x rotation angle of this **ccAffineRectangle**. The x rotation angle is the angle between the Po-Px side of the affine rectangle and the client coordinate system x-axis.

  Notes

  The value returned by this function is between $-\pi$ and π radians.

- `void xRotation (const ccRadian& xRot);`

  Sets the x rotation angle of this **ccAffineRectangle**. The x rotation angle is the angle between the Po-Px side of the affine rectangle and the client coordinate system x-axis.

  The center point, height, and width of the **ccAffineRectangle** are not affected.

  Parameters

  xRot The angle to set.

yRotation

```
ccRadian yRotation () const;
```

Returns the y rotation angle of this **ccAffineRectangle**. The y rotation angle is the angle between the Po-Py side of affine rectangle and the client coordinate system y-axis.

Notes

The value returned by this function is between $-\pi$ and π radians.

skew

```
ccRadian skew () const;
void skew (const ccRadian& skew);
```

- `ccRadian skew () const;`

  Returns the skew angle of this **ccAffineRectangle**. The skew angle is the angle between a line drawn perpendicular to the Po-Px side of the affine rectangle and the Po-Py side of the rectangle. The skew angle is equal to the y rotation angle minus the x rotation angle.

  Notes

  The value returned by this function is between $-\pi/2$ and $\pi/2$ radians.

- `void skew (const ccRadian& skew);`

  Sets the skew angle of this **ccAffineRectangle**. The skew angle is the angle between a line drawn perpendicular to the Po-Px side of the affine rectangle and the Po-Py side of the rectangle. The skew angle is equal to the y rotation angle minus the x rotation angle.

  The center point, height, and width of the **ccAffineRectangle** are not affected.

  Parameters

  skew The skew angle

  Throws

  ccAffineRectangle::BadSkew

  skew is $\pi/2$ radians or $-\pi/2$ radians (These two values are illegal; they produce a degenerate affine rectangle.) Note that this error is thrown if *skew* is within **ckAffineRectangleSkewAngleTol** of $\pi/2$ radians or $-\pi/2$ radians. You can determine if a given skew angle is valid by calling **ccAffineRectangle::isBadSkew()**.

cornersPoPxPy

```
void cornersPoPxPy (const cc2Vect& cornerPo,
  const cc2Vect& cornerPx, const cc2Vect& cornerPy);
```

Sets the locations of the Po, Px, and Py vertices of this **ccAffineRectangle**.

Parameters

cornerPo The location of the Po vertex, in client coordinates.

cornerPx The location of the Px vertex, in client coordinates.

cornerPy The location of the Py vertex, in client coordinates.

Notes

This function fully defines an affine rectangle.

centerLengthsRotAndSkew

```
void centerLengthsRotAndSkew (const cc2Vect& center,
  double xLen, double yLen,
  const ccRadian& xRotation = ccRadian(0),
  const ccRadian& skew = ccRadian(0));
```

Sets the location, width, height, rotation angle, and skew angle of this **ccAffineRectangle**.

Parameters

center The center of the **ccAffineRectangle**, in client coordinates.

xLen The length of the Po-Px side of the **ccAffineRectangle**, in client coordinates.

yLen The length of the Po-Py side of the **ccAffineRectangle**, in client coordinates.

xRotation The rotation angle of the **ccAffineRectangle**.

skew The skew angle of the **ccAffineRectangle**.

Throws

ccAffineRectangle::BadLength
xLen or *yLen* is less than or equal to 0.

ccAffineRectangle::BadSkew
skew is $\pi/2$ radians or $-\pi/2$ radians (These two values are illegal; they produce a degenerate affine rectangle.) Note that this error is thrown if *skew* is within **ckAffineRectangleSkewAngleTol** of $\pi/2$ radians or $-\pi/2$ radians. You can determine if a given skew angle is valid by calling **ccAffineRectangle::isBadSkew()**.

Notes

This function fully defines an affine rectangle.

cornerPoLengthsRotAndSkew

```
void cornerPoLengthsRotAndSkew (const cc2Vect& cornerPo,
  double xLen, double yLen,
  const ccRadian& xRotation = ccRadian(0),
  const ccRadian& skew = ccRadian(0));
```

Sets the location of the Po vertex and the width, height, rotation angle, and skew angle of this **ccAffineRectangle**.

Parameters

cornerPo The location of the Po vertex, in client coordinates.

xLen The length of the Po-Px side of the **ccAffineRectangle**, in client coordinates.

yLen The length of the Po-Py side of the **ccAffineRectangle**, in client coordinates.

xRotation The rotation angle of the **ccAffineRectangle**.

skew The skew angle of the **ccAffineRectangle**.

Throws

ccAffineRectangle::BadLength
xLen or *yLen* is less than or equal to 0.

ccAffineRectangle::BadSkew
skew is π/2 radians or -π/2 radians (These two values are illegal; they produce a degenerate affine rectangle.) Note that this error is thrown if *skew* is within **ckAffineRectangleSkewAngleTol** of π/2 radians or -π/2 radians. You can determine if a given skew angle is valid by calling **ccAffineRectangle::isBadSkew()**.

Notes

This function fully defines an affine rectangle.

map

```
ccAffineRectangle map(const cc2Xform& c) const;
```

Maps this **ccAffineRectangle** by the supplied **cc2Xform** and returns the result. None of the dimensions of this **ccAffineRectangle** change. Useful for scaling.

Parameters

c The **cc2Xform** by which to map this **ccAffineRectangle**.

encloseImageRect

```
ccPelRect encloseImageRect(
  const cc2Xform& imageFromClientXform = cc2Xform::I)
  const;
```

Returns the smallest rectangle aligned to the supplied coordinate system that encloses this **ccAffineRectangle**. By default, this function returns the smallest enclosing rectangle in the image coordinate system.

Parameters

imageFromClientXform
A **cc2Xform** that specifies the transformation from the image coordinate system. A default parameter of the identity transformation is supplied.

degen

```
bool degen() const;
```

Returns true if this **ccAffineRectangle** is degenerate. A **ccAffineRectangle** is considered degenerate if either its width or height is 0.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this affine rectangle.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For affine rectangles, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

For affine rectangles, this function always returns true. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For affine rectangles, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For affine rectangles, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

Returns true for most affine rectangles. Returns false if all four vertices are coincident, in which case the affine rectangle collapses to a single point and there is no tangent. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For affine rectangles, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For affine rectangles, this function always returns true. See **ccShape::reverse()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this affine rectangle. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this affine rectangle. See **ccShape::boundingBox()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

Returns true if this affine rectangle is right-handed. See **ccShape::isRightHanded()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this affine rectangle to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

sample

```
virtual void sample(const ccSampleParams &params,
  ccSampleResult &result);
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores affine rectangles for which **hasTangent()** is false.

Throws

ccShapesError::SampleOverflow
Supplied spacing and tolerance bounds require more than *maxPoint* samples to be generated. See **ccSampleParams** for details.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this affine rectangle mapped by *X*.

Parameters

X The transformation object.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccLineSeg**s. See **ccShape::decompose()** for additional information.

within

```
virtual bool within(const cc2Vect& v) const;
```

Returns true if the given point is within this affine rectangle.

Parameters

v The point.

See **ccShape::within()** for more information.

Static Functions

isBadSkew

```
static bool isBadSkew(const ccRadian& skew);
```

Returns true if the supplied value is within **ckAffineRectangleSkewAngleTol** radians of $\pi/2$ radians or $-\pi/2$ radians. The function returns false otherwise.

Parameters

skew The skew angle to test.

Deprecated Members

encloseRect `ccRect encloseRect() const;`

Use **boundingBox()** instead.

distToPoint `double distToPoint(const cc2Vect& v) const;`

Use **distanceToPoint()** (inherited from the **ccShape** base class) instead.

ccAffineSamplingParams

```
#include <ch_cvl/affsampl.h>

class ccAffineSamplingParams;
```

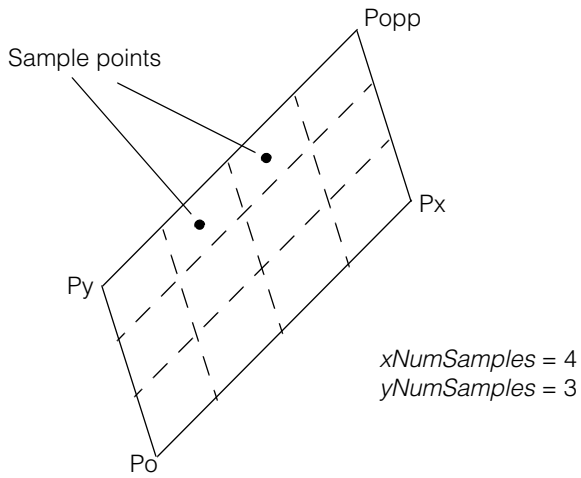
Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class describes affine sampling parameters. Affine sampling parameters control how the pixels within an affine rectangle in an image are sampled. The following are affine sampling parameters:

- The number of x-axis and y-axis divisions within an affine rectangle
- The sampling method
- An affine rectangle

See the following diagram.



Constructors/Destructors

ccAffineSamplingParams

```
ccAffineSamplingParams ();

ccAffineSamplingParams (
  const ccAffineRectangle& affRect,
  c_Int32 xNumSamples,
  c_Int32 yNumSamples,
  Interpolation method = kDefaultInterpolation);

ccAffineSamplingParams(const ccAffineSamplingParams&);
```

- `ccAffineSamplingParams ();`

  Default constructor.

- ```
  ccAffineSamplingParams (
    const ccAffineRectangle& affRect,
    c_Int32 xNumSamples,
    c_Int32 yNumSamples,
    Interpolation method = kDefaultInterpolation);
  ```

  Constructs a **ccAffineSamplingParams** using the supplied values.

  Parameters

  affRect — The **ccAffineRectangle** that defines the rectangle to sample

  xNumSamples — The number of divisions into which *affrect* is divided along the Po-Px side.

  yNumSamples — The number of divisions into which *affrect* is divided along the Po-Py side.

  method — The interpolation method to use to compute the sampled pixel values. *method* must be one of the following values:

  ccAffineSamplingParams::eNone
  ccAffineSamplingParams::eBilinear
  ccAffineSamplingParams::eHighPrecision

  Throws

  ccAffineSamplingParams::BadNumSamples
  xNumSamples or *yNumSamples* is less than 1.
  ```

- `ccAffineSamplingParams(const ccAffineSamplingParams&);`

  Copy constructor.

Enumerations

Interpolation `enum Interpolation`

This enumeration defines the sampling methods supported by the tool.

Value	Meaning
eNone = 1	Use nearest-neighbor interpolation.
eBilinear = 2 (Also called *eBilinearApprox*)	Use bilinear interpolation of four nearest pixels. This interpolation mode approximates true bilinear interpolation. While this is the default mode (for backward compatibility), Cognex recommends using *eBilinearAccurate*.
eHighPrecision = 3	Use high-precision interpolation to compute the interpolated value.
eBilinearAccurate = 4	Performs an accurate bilinear interpolation of the four nearest pixels. This method is the most accurate, and in most cases, the fastest method.
kDefaultInterpolation = *eBilinear*	The default interpolation.

Operators

operator=

```
ccAffineSamplingParams& operator=(
                          const ccAffineSamplingParams& rhs);
```

Assignment operator. Make this object a copy of *rhs*.

Parameters

rhs The copy source.

operator==

```
bool operator==(const ccAffineSamplingParams& that);
```

Returns true if this object is equal to *that*. Returns false otherwise.

Two **ccAffineSamplingParams** objects are equal if their affine rectangles are the same and the number of samples and interpolation methods are equal.

Parameters

that The other **ccAffineSamplingParams** object.

Public Member Functions

affineRectangle

```
const ccAffineRectangle& affineRectangle () const;
void affineRectangle (const ccAffineRectangle& affRect);
```

- ```
  const ccAffineRectangle& affineRectangle () const;
  ```

  Returns a reference to the **ccAffineRectangle** associated with this **ccAffineSamplingParams**.

- ```
  void affineRectangle (const ccAffineRectangle& affRect);
  ```

  Sets the **ccAffineRectangle** associated with this **ccAffineSamplingParams**.

  Parameters

  affRect The **ccAffineRectangle** to set.

xNumSamples

```
c_Int16 xNumSamples () const;
void xNumSamples (c_Int32 xNum);
```

- ```
  c_Int16 xNumSamples () const;
  ```

  This overload is deprecated. Use **xNumSamples32()** to obtain the current number of Po-Px divisions in this **ccAffineSamplingParams**.

  Throws

  ccAffineSamplingParams:BadNumSamples
  The number of samples (set using **xNumSamples32**) is greater than 32767.
  ```

- `void xNumSamples (c_Int32 xNum);`

  Sets the number of Po-Px divisions in this **ccAffineSamplingParams**. The **ccAffineRectangle** associated with this **ccAffineSamplingParams** is divided into the specified number of divisions along the Po-Px side.

  Parameters
  xNum The number of x-divisions

  Throws
  ccAffineSamplingParams:BadNumSamples
  xNum is less than 1.

yNumSamples

`c_Int16 yNumSamples () const;`

`void yNumSamples (c_Int32 yNum);`

- `c_Int16 yNumSamples () const;`

  This overload is deprecated. Use **yNumSamples32()** to obtain the current number of Po-Py divisions in this **ccAffineSamplingParams**.

  Throws
  ccAffineSamplingParams:BadNumSamples
  The number of samples (set using **yNumSamples32**) is greater than 32767.

- `void yNumSamples (c_Int32 yNum);`

  Sets the number of Po-Py divisions in this **ccAffineSamplingParams**. The **ccAffineRectangle** associated with this **ccAffineSamplingParams** is divided into the specified number of divisions along the Po-Py side.

  Parameters
  yNum The number of y-divisions

  Throws
  ccAffineSamplingParams:BadNumSamples
  yNum is less than 1.

xNumSamples32 `c_Int32 xNumSamples32 () const;`

Returns the number of Po-Px divisions in this **ccAffineSamplingParams**. The **ccAffineRectangle** associated with this **ccAffineSamplingParams** is divided into the returned number of divisions along the Po-Px side.

yNumSamples32 `c_Int32 yNumSamples32 () const;`

Returns the number of Po-Py divisions in this **ccAffineSamplingParams**. The **ccAffineRectangle** associated with this **ccAffineSamplingParams** is divided into the returned number of divisions along the Po-Py side.

interpolation

```
Interpolation interpolation () const;
void interpolation (Interpolation method);
```

- `Interpolation interpolation () const;`

  Returns the interpolation method of this **ccAffineSamplingParams**. The returned value is one of the following:

  ccAffineSamplingParams::eNone
  ccAffineSamplingParams::eBilinear
  ccAffineSamplingParams::eBilinearApprox
  ccAffineSamplingParams::eBilinearAccurate
  ccAffineSamplingParams::eHighPrecision

- `void interpolation (Interpolation method);`

  Sets the interpolation method of this **ccAffineSamplingParams**.

  Parameters

  method — The interpolation method to use. *method* must be one of the following values:

  ccAffineSamplingParams::eNone
  ccAffineSamplingParams::eBilinear
  ccAffineSamplingParams::eBilinearApprox
  ccAffineSamplingParams::eBilinearAccurate
  ccAffineSamplingParams::eHighPrecision

willClip `bool willClip(const cc_PelBuffer& srcImg) const;`

Returns true if the affine rectangle specified in this **ccAffineSamplingParams** would need to access pixels outside of the valid region of *srcImg*. The valid region within *srcImg* is defined as being all pixels in *srcImg* other than a two-pixel wide border around the edge of *srcImg*.

Parameters

srcImage — The image to test.

Throws

ccAffineSamplingParams::NotImplemented if the interpolation

ccAffProjImgDefs::NotImplemented

interpolation() is *ccAffineSamplingParams::eBilinearApprox* or *ccAffineSamplingParams::eHighPrecision* and *srcImg.rowUpdate()* or *srcImg.height()* is greater than or equal to 32768.

ccAffineSamplingParams

ccAnalogAcqProps

```
#include <ch_cvl/prop.h>

class ccAnalogAcqProps : public ccAcqProps;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class is has been deprecated. Use the class **ccAcqProps** instead. See **ccAcqProps** on page 253.

ccAnalogAcqProps

ccAngle8

```
#include <ch_cvl/units.h>

class ccAngle8;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes an angle as an unsigned 8-bit integer with a value from 0 to 255. Each integer value represents a small range of angles:

Integer value	Angle in degrees	Range
0	0°	±0.703125
1	1.40625°	±0.703125
2	2.8125°	±0.703125
32	45.0°	±0.703125

Note that you can use the constructors to convert from one angle representation to another. For example, to specify 60° as a **ccAngle8**, you could write:

```
ccAngle8 ang8(ccDegree(60.0));
```

When converting angles, be aware that Cognex vision tools are not entirely consistent in their use of **ccAngle8** and **ccAngle16**. Some tools use a truncation model when converting between floating point angles and binary angles, and other tools use a rounding model. The following table shows an example of the different results produced when converting angles between floating point and **ccAngle8** representations using the truncation method as compared to the rounding method:

Convert From	Convert To	Truncation	Rounding
1.406°	**ccAngle8**	0	1
-0.001°	**ccAngle8**	255	0
0 **ccAngle8**	degrees	0°	0.703125°
1 **ccAngle8**	degrees	1.40625°	2.109375°

The rounding method is numerically more stable if multiple conversions will be made between floating point and binary angles. For example, using the truncation model a floating point computation that yields -0.0000000001° is represented as a 255 **ccAngle8**, which is then converted back to 358.59375°. If an iterative computation then added another -0.0000000001° and converted back to **ccAngle8**, you would get a steadily decreasing representation of 254, 253, 252, and so on.

Constructors/Destructors

ccAngle8

```
ccAngle8();
ccAngle8(ccRadian a);
ccAngle8(ccDegree a);
ccAngle8(ccAngle16 a);
explicit ccAngle8(c_UInt8 a);
```

- `ccAngle8();`

  Creates an uninitialized **ccAngle8** object.

- `ccAngle8(ccRadian a);`

  Creates a **ccAngle8** object from the given **ccRadian** object.

  Parameters

  a — The **ccRadian** object to convert to **ccAngle8** format.

- `ccAngle8(ccDegree a);`

  Creates a **ccAngle8** object from the given **ccDegree** object.

  Parameters

  a — The **ccDegree** object to convert.

- `ccAngle8(ccAngle16 a);`

  Creates a **ccAngle8** object from the given **ccAngle16** object.

  Parameters

  a — The **ccAngle16** object to convert.

- `explicit ccAngle8(c_UInt8 a);`

  Creates a **ccAngle8** object with the specified value.

  Parameters

  a The angle as an unsigned 8-bit integer.

Operators

operator+

```
ccAngle8 operator+(ccAngle8 a) const;
```

Returns the result of adding the angle *a* to this angle.

Parameters

a The angle to add to this angle.

---

operator-

```
ccAngle8 operator-(ccAngle8 a) const;
ccAngle8 operator-() const;
```

- `ccAngle8 operator-(ccAngle8 a) const;`

  Returns the result of subtracting the angle *a* from this angle.

  Parameters

  a The angle to subtract from this angle.

- `ccAngle8 operator-() const;`

  Returns the negative of this angle. The unary minus operator.

---

operator*

```
ccAngle8 operator*(c_UInt8 a) const;
c_UInt8 operator*(ccAngle8 a) const;
friend ccAngle8 operator*(c_UInt8 a, ccAngle8 b);
```

- `ccAngle8 operator*(c_UInt8 a) const;`

  Returns the result of multiplying this angle by *a*.

  Parameters

  a The amount to multiply by.

- `c_UInt8 operator*(ccAngle8 a) const;`

  Returns the result of multiplying this angle by the angle *a*.

  Parameters

a	The angle to multiply by.

- `friend ccAngle8 operator*(c_UInt8 a, ccAngle8 b);`

  Returns the result of multiplying the angle *b* by *a*.

  Parameters

a	The amount to multiply by.
b	The angle to multiply.

operator/

```
ccAngle8 operator/(c_UInt8 a) const;
c_UInt8 operator/(ccAngle8 a) const;
```

- `ccAngle8 operator/(c_UInt8 a) const;`

  Returns the result of dividing this angle by *a*.

  Parameters

a	The amount to divide by.

- `c_UInt8 operator/(ccAngle8 a) const;`

  Returns the result of dividing this angle by the angle *a*.

  Parameters

a	The angle to divide by.

operator= `ccAngle8& operator=(c_UInt8 a);`

Assigns the value *a* to this angle.

Parameters

a	The value to assign.

operator*=

```
ccAngle8& operator*=(c_UInt8 a);
ccAngle8& operator*=(ccAngle8 a);
```

- `ccAngle8& operator*=(c_UInt8 a);`

  Multiplies this angle by *a* and returns the result.

  Parameters

a	The value to multiply by.

- `ccAngle8& operator*=(ccAngle8 a);`

  Multiplies this angle by the angle *a* and returns the result.

  Parameters

a	The angle to multiply by.

operator/=

```
ccAngle8& operator/=(c_UInt8 a);
ccAngle8& operator/=(ccAngle8 a);
```

- `ccAngle8& operator/=(c_UInt8 a);`

  Divides this angle by the value *a* and returns the result.

  Parameters

a	The value to divide by.

- `ccAngle8& operator/=(ccAngle8 a);`

  Divides this angle by the angle *a* and returns the result.

  Parameters

a	The angle to divide by.

operator+=

```
ccAngle8& operator+=(ccAngle8 a);
```

Adds the angle *a* to this angle and returns the result.

Parameters

a	The angle to add.

operator-= `ccAngle8& operator-=(ccAngle8 a);`

Subtracts the angle *a* from this angle and returns the result.

Parameters

a The angle to subtract.

operator== `bool operator!=(ccAngle8 a) const;`

Returns true if this angle is exactly equal to the angle *a*.

Parameters

a The other angle.

operator!= `bool operator!=(ccAngle8 a) const;`

Returns true if this angle is not equal to the angle *a*.

Parameters

a The other angle.

operator< `bool operator<(ccAngle8 a) const;`

Returns true if this angle is less than angle *a*.

Parameters

a The other angle.

operator<= `bool operator<=(ccAngle8 a) const;`

Returns true if this angle is less than or equal to angle *a*.

Parameters

a The other angle.

operator> `bool operator>(ccAngle8 a) const;`

Returns true if this angle is greater than angle *a*.

Parameters

a The other angle.

operator>= `bool operator>=(ccAngle8 a) const;`

Returns true if this angle is greater than or equal to angle *a*.

Parameters

a	The other angle.

Public Member Functions

toDouble

```
double toDouble() const;
```

Returns this angle as a **double**.

plain

```
c_UInt8 plain() const;
```

Returns this angle as a **c_UInt8**.

ccAngle8

ccAngle16

```
#include <ch_cvl/units.h>

class ccAngle16;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes an angle as an unsigned 16-bit integer with a value from 0 to 65535. Each integer value represents a small range of angles:

Integer value	Angle in degrees	Range
0	0°	±0.00274658
1	0.00549316°	±0.00274658
2	0.010986°	±0.00274658
8192	45.0°	±0.00274658

Note that you can use the constructors to convert from one angle representation to another. For example, to specify 60° as a **ccAngle16**, you could write:

```
ccAngle16 ang16(ccDegree(60.0));
```

When converting angles, be aware that Cognex vision tools are not entirely consistent in their use of **ccAngle8** and **ccAngle16**. Some tools use a truncation model when converting between floating point angles and binary angles, and other tools use a rounding model. The following table shows an example of the different results produced when converting angles between floating point and **ccAngle16** representations using the truncation method as compared to the rounding method:

Convert From	Convert To	Truncation	Rounding
0.00549°	**ccAngle16**	0	1
-0.001°	**ccAngle16**	65535	0
0 **ccAngle16**	degrees	0°	0.00274658203125°
1 **ccAngle16**	degrees	0.0054931640625°	0.00823974609375°

The rounding method is numerically more stable if multiple conversions will be made between floating point and binary angles. For example, using the truncation model a floating point computation that yields -0.0000000001° is represented as a 65535 **ccAngle16**, which is then converted back to 359.9945068360375°. If an iterative computation then added another -0.0000000001° and converted back to **ccAngle16**, you would get a steadily decreasing representation of 65534, 65533, 65532, and so on.

Constructors/Destructors

ccAngle16

```
ccAngle16();
ccAngle16(ccRadian a);
ccAngle16(ccDegree a);
ccAngle16(ccAngle8 a);
explicit ccAngle16(c_UInt16 a);
```

- `ccAngle16();`

  Creates an uninitialized **ccAngle16** object.

- `ccAngle16(ccRadian a);`

  Creates a **ccAngle16** object from the given **ccRadian** object.

  Parameters

  a The **ccRadian** object to convert to degrees.

- `ccAngle16(ccDegree a);`

  Creates a **ccAngle16** object from the given **ccDegree** object.

  Parameters

  a The **ccDegree** object to convert.

- `ccAngle16(ccAngle8 a);`

  Creates a **ccAngle16** object from the given **ccAngle8** object.

  Parameters

  a The **ccAngle8** object to convert.

- `explicit ccAngle16(c_UInt16 a);`

  Creates a **ccAngle16** object with the specified value.

  Parameters

a	The angle as an unsigned 16-bit integer.

Operators

operator+

```
ccAngle16 operator+(ccAngle16 a) const;
```

Returns the result of adding the angle *a* to this angle.

Parameters

a	The angle to add to this angle.

operator-

```
ccAngle16 operator-(ccAngle16 a) const;
ccAngle16 operator-() const;
```

- `ccAngle16 operator-(ccAngle16 a) const;`

  Returns the result of subtracting the angle *a* from this angle.

  Parameters

a	The angle to subtract from this angle.

- `ccAngle16 operator-() const;`

  Returns the negative of this angle. The unary minus operator.

operator*

```
ccAngle16 operator*(c_UInt16 a) const;
c_UInt16 operator*(ccAngle16 a) const;
friend ccAngle16 operator*(c_UInt16 a, ccAngle16 b);
```

- `ccAngle16 operator*(c_UInt16 a) const;`

  Returns the result of multiplying this angle by *a*.

  Parameters

a	The amount to multiply by.

- `c_UInt16 operator*(ccAngle16 a) const;`

  Returns the result of multiplying this angle by the angle *a*.

  Parameters

a	The angle to multiply by.

- `friend ccAngle16 operator*(c_UInt16 a, ccAngle16 b);`

  Returns the result of multiplying the angle *b* by *a*.

  Parameters

a	The amount to multiply by.
b	The angle to multiply.

operator/

```
ccAngle16 operator/(c_UInt16 a) const;
c_UInt16 operator/(ccAngle16 a) const;
```

- `ccAngle16 operator/(c_UInt16 a) const;`

  Returns the result of dividing this angle by *a*.

  Parameters

a	The amount to divide by.

- `c_UInt16 operator/(ccAngle16 a) const;`

  Returns the result of dividing this angle by the angle *a*.

  Parameters

a	The angle to divide by.

operator= `ccAngle16& operator=(c_UInt16 a);`

Assigns the value *a* to this angle.

Parameters

a	The value to assign.

operator*=

```
ccAngle16& operator*=(c_UInt16 a);
ccAngle16& operator*=(ccAngle16 a);
```

- `ccAngle16& operator*=(c_UInt16 a);`

  Multiplies this angle by *a* and returns the result.

  Parameters

a	The value to multiply by.

- `ccAngle16& operator*=(ccAngle16 a);`

  Multiplies this angle by the angle *a* and returns the result.

  Parameters

a	The angle to multiply by.

operator/=

```
ccAngle16& operator/=(c_UInt16 a);
ccAngle16& operator/=(ccAngle16 a);
```

- `ccAngle16& operator/=(c_UInt16 a);`

  Divides this angle by the value *a* and returns the result.

  Parameters

a	The value to divide by.

- `ccAngle16& operator/=(ccAngle16 a);`

  Divides this angle by the angle *a* and returns the result.

  Parameters

a	The angle to divide by.

operator+=

```
ccAngle16& operator+=(ccAngle16 a);
```

Adds the angle *a* to this angle and returns the result.

Parameters

a	The angle to add.

operator-= `ccAngle16& operator-=(ccAngle16 a);`

Subtracts the angle *a* from this angle and returns the result.

Parameters

a The angle to subtract.

operator== `bool operator!=(ccAngle16 a) const;`

Returns true if this angle is exactly equal to the angle *a*.

Parameters

a The other angle.

operator!= `bool operator!=(ccAngle16 a) const;`

Returns true if this angle is not equal to the angle *a*.

Parameters

a The other angle.

operator< `bool operator<(ccAngle16 a) const;`

Returns true if this angle is less than angle *a*.

Parameters

a The other angle.

operator<= `bool operator<=(ccAngle16 a) const;`

Returns true if this angle is less than or equal to angle *a*.

Parameters

a The other angle.

operator> `bool operator>(ccAngle16 a) const;`

Returns true if this angle is greater than angle *a*.

Parameters

a The other angle.

operator>= `bool operator>=(ccAngle16 a) const;`

Returns true if this angle is greater than or equal to angle *a*.

Parameters

a	The other angle.

Public Member Functions

toDouble

```
double toDouble() const;
```

Returns this angle as a **double** value.

plain

```
c_UInt16 plain() const;
```

Returns this angle as a **c_UInt16**.

ccAngle16

ccAngleRange

```
#include <ch_cvl/range.h>

class ccAngleRange;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

The class **ccAngleRange** specifies a range of orientations and provides methods for performing the following operations:

- Check whether a value is within a range of angles.
- Check whether a range of angles is full, partly full, or empty.
- Dilate ranges of angles (Minkowski sum operation).

This class is can specify a range that contains an angle whose measure is 2 * π radians. For example, an angle range from 6.0 to 7.0 radians would contain the angle of 0.0 degrees (since 6.28 is within the range [6.0 through 7.0] and 0 ~= 6.28 (2 * π).

Constructors/Destructors

ccAngleRange

```
ccAngleRange ();

ccAngleRange (const ccRadian &start, const ccRadian &end);
```

- `ccAngleRange ();`

  Returns a default constructed empty angle range.

- `ccAngleRange (const ccRadian &start, const ccRadian &end);`

  Returns an **ccAngleRange** object characterized by the specified start and end values. If (**end.toDouble()** - **start.toDouble()**) > 2 * π, then the function constructs a full range. Otherwise, it constructs a partial range.

  Parameters

start	The start position of the angle range.
end	The end position of the angle range.

Operators

operator==

```
bool operator== (const ccAngleRange& that) const;
```

True if this range equals the other range; false otherwise.

Parameters

that The other range.

Notes

Two **ccAngleRange** objects are considered equal if their ranges, start angles, and end angles are equal.

operator!=

```
bool operator!= (const ccAngleRange& that) const;
```

True if this range does not equal that range; false if this range equals that range.

Parameters

that The other range.

Notes

Two **ccAngleRange** objects are considered equal if their ranges, start angles, and end angles are equal or if they are both full or both empty.

Public Member Functions

dilate

```
ccAngleRange dilate(const ccAngleRange &x) const;
```

Returns a new **ccAngleRange** object that is created by dilating this range with another range specified by *x*. Dilation is accomplished by using the Minkowski sum of two sets A and B. The resulting range is the set of all points which can be generated by adding an element of A to an element of B.

Parameters

x The range used to dilate this angle range.

end

```
ccRadian end() const;
```

Returns the end position of a partial range.

Throws

ccRangeDefs::NotPartialRange
This is not a partial angle range.

isWithin

```
bool isWithin(const ccRadian &x) const;
```

Returns a bool that indicates whether or not the value *x* is inside this angle range.

Parameters

x The value that may or may not be inside this angle range.

length

```
ccRadian length() const;
```

Returns the length of the angle range which is defined as one of the following:

ck2Pi if the angle range is full (range_ == *eFull*)

0 if the angle range is empty (range_ == *eEmpty*)

*end_-start_* if the angle range contains some members (range_ == ePartial).

middle

```
ccRadian middle() const;
```

Returns the middle of the given angle range.

Throws

ccRangeDefs::NotPartialRange
This is not a partial angle range.

start

```
ccRadian start() const;
```

Returns the start position of a partial angle range.

Throws

ccRangeDefs::NotPartialRange
This is not a partial angle range.

Static Functions

EmptyAngleRange

```
static ccAngleRange EmptyAngleRange();
```

Returns the default constructed empty angle range.

FullAngleRange

```
static ccAngleRange FullAngleRange();
```

Returns the default constructed full angle range.

ccAngleRange

ccAnnulus

```
#include <ch_cvl/shapes.h>

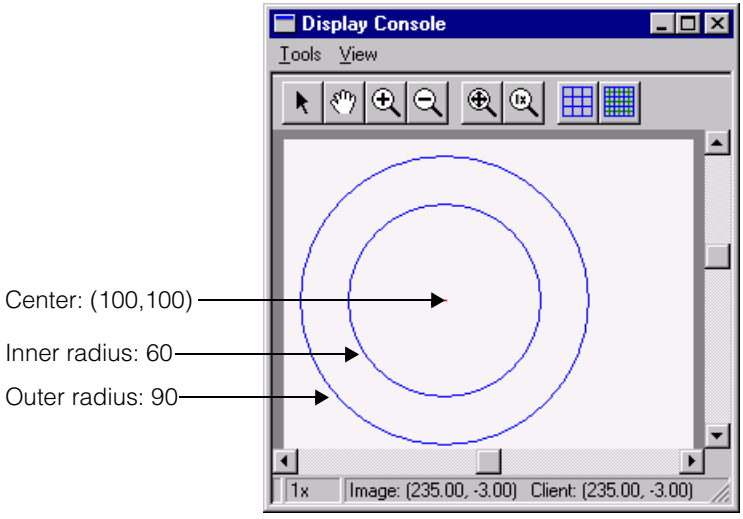
class ccAnnulus : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccAnnulus** class describes an annulus, a ring shape made up of two concentric circles. You specify a center and the inner and outer radii to describe an annulus. The following figure shows an annulus.

```
ccAnnulus(cc2Vect(100,100), 60, 90)
```



Constructors/Destructors

ccAnnulus

```
ccAnnulus();
ccAnnulus(const cc2Vect& c, double r1, double r2);
ccAnnulus(const ccCircle& c1, const ccCircle& c2);
```

- `ccAnnulus();`

  The default constructor creates a degenerate annulus with its center at (0,0) and inner and outer radii set to zero.

- `ccAnnulus(const cc2Vect& c, double r1, double r2);`

  Creates an annulus with the specified center and radii.

  Parameters

  c The center of the annulus.

  r1 The inner radius. The radius must be non-negative.

  r2 The outer radius. The radius must be non-negative.

  Throws

  ccShapesError::BadRadius
  Either radius is negative.

- `ccAnnulus(const ccCircle& c1, const ccCircle& c2);`

  Creates an annulus from two concentric circles.

  Parameters

  c1 The inner circle.

  c2 The outer circle.

  Throws

  ccShapesError::NotConcentric
  Circles c1 and c2 are not concentric.

Operators

operator==

```
bool operator==(const ccAnnulus& other) const;
```

Returns true if this annulus is equal to another annulus: it has the same center, inner radius, and outer radius as the other annulus.

Parameters

other	The other annulus.

operator!=

```
bool operator!=(const ccAnnulus&) const;
```

Returns true if this annulus is not equal to another annulus.

Parameters

other	The other annulus.

Public Member Functions

center

```
const ccPoint& center() const;
```

Returns the center of the annulus.

innerRadius

```
double innerRadius() const;
```

Returns the inner radius of the annulus.

outerRadius

```
double outerRadius() const;
```

Returns the outer radius of the annulus.

innerCircle

```
ccCircle innerCircle() const;
```

Returns the inner circle of the annulus.

outerCircle

```
ccCircle outerCircle() const;
```

Returns the outer circle of the annulus.

map

```
ccGenAnnulus map(const cc2Xform& c) const;
```

Returns an annulus that is the result of mapping this annulus with the transformation object *c*.

Parameters

c The transformation object.

Notes

Both of the radii of this annulus must be strictly positive, and the transform *c* must be nonsingular. If both of these conditions are false, use **mapshape()** instead.

degen

```
bool degen() const;
```

Returns true if this annulus is *degenerate*. An annulus is degenerate if either both radii are equal, or one of the radii is less than or equal to zero.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this annulus.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For annuli, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

For annuli, this function always returns true, even for annuli that are degenerate. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For annuli, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For annuli, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

This function returns true if at least one of the radii of this annulus is positive. It returns false if neither radius is positive. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For annuli, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For annuli, this function always returns false. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this annulus. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this annulus to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this annulus to the given point, as determined by **nearestPoint()**.

Parameters

info	Shape information for this annulus.
point	The point.

See **ccShape::nearestPerimPos()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

For annuli, this function always returns true. Annuli are always right-handed. See **ccShape::isRightHanded()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params Specifies details of how the sampling should be done.

result Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores annuli for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this annulus mapped by *X*.

Parameters

X The transformation object.

Notes

If *X* is the identity transform, this function returns a **ccAnnulus**. Otherwise, if *X* is singular this function returns a **ccRegionTree**, or if *X* is nonsingular and either radius is zero, this function returns a **ccEllipseAnnulus**. In all other cases, it returns a **ccGenAnnulus**.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccEllipseArc2**s. See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect& v) const;
```

Returns true if the given point is within this annulus.

Parameters

v The point.

Notes

A point is within an annulus if it is between the inner and outer circles of the annulus.

subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this annulus over the given perimeter range. The perimeter of the final returned shape is equal to the absolute value of the distance component of *range*, assuming the distance is not clipped.

Parameters

info Shape information for this annulus.

range The perimeter range.

See **ccShape::subShape()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

ccAnnulus

ccArchive

```
#include <ch_cvl/archive.h>

class ccArchive;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Not applicable

The **ccArchive** class is an abstract base class from which the concrete archive classes **ccFileArchive** and **ccMemoryArchive** are derived. This class implements the **||**, **<<**, and **>>** operators for the built-in C++ types.

Constructors/Destructors

ccArchive

```
virtual ~ccArchive();
```

Enumerations

Direction

```
enum Direction;
```

This enumeration defines the mode for a **ccArchive**.

Value	Meaning
eLoading	This archive is used to load objects.
eStoring	This archive is used to store objects.

Ordering

```
enum Ordering;
```

This enumeration defines the endianness of a **ccArchive**.

The endianness of an archive is the endianness with which data are stored in the archive. Whenever an archive is loaded, if the endianness of the machine onto which it is being loaded does not match the endianness of the archive, byte swapping is performed as the objects are loaded.

When you instantiate a **ccArchive**-derived concrete class for storing, you must specify the endianness of the archive. If the endianness of the archive does not match the endianness of the machine from which the objects are being stored, then byte swapping is performed as the objects are stored.

The constructors for all **ccArchive**-derived concrete classes provide a default value of *ccArchive::eLittleEndian* for the endianness of the archive.

Value	Meaning
eBigEndian	Big-endian.
eLittleEndian	Little-endian.

Operators

operatorII

```
ccArchive& operator|| (c_UInt8& val);
ccArchive& operator|| (c_Int8& val);
ccArchive& operator|| (char& val);
ccArchive& operator|| (c_UInt16& val);
ccArchive& operator|| (c_Int16& val);
ccArchive& operator|| (c_UInt32& val);
ccArchive& operator|| (c_Int32& val);
ccArchive& operator|| (c_UInt64& val);
ccArchive& operator|| (c_Int64& val);
ccArchive& operator|| (unsigned int& val);
ccArchive& operator|| (int& val);
ccArchive& operator|| (float& val);
ccArchive& operator|| (double& val);
ccArchive& operator|| (cmStd string& val);
ccArchive& operator|| (cmStd wstring& val);
ccArchive& operator|| (wchar_t& val);
ccArchive& operator|| (ccCvlString32& val);
ccArchive& operator|| (bool& val);
```

Notes

All of the overloads of `operator||` may throw the following errors:

ccArchive::BadType
: The stream data do not match the expected type while loading (the stream position after this error is unspecified).

ccArchive::Eof
: There is no more input data (loading only).

ccArchive::BadBuffer
: The archive is broken and unusable. If this error occurs during storing, it may indicate that the output device (disk) is full.

- `ccArchive& operator|| (c_UInt8& val);`

  Serializes the supplied **c_UInt8** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters
  val The value to serialize.

- `ccArchive& operator|| (c_Int8& val);`

  Serializes the supplied **c_Int8** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters
  val The value to serialize.

- `ccArchive& operator|| (char& val);`

  Serializes the supplied **char** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters
  val The value to serialize.

- `ccArchive& operator|| (c_UInt16& val);`

  Serializes the supplied **c_UInt16** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters
  val The value to serialize.

- `ccArchive& operator|| (c_Int16& val);`

  Serializes the supplied **c_Int16** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters
  val The value to serialize.

- `ccArchive& operator|| (c_UInt32& val);`

  Serializes the supplied **c_UInt32** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val The value to serialize.

- `ccArchive& operator|| (c_Int32& val);`

  Serializes the supplied **c_Int32** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val The value to serialize.

- `ccArchive& operator|| (c_UInt64& val);`

  Serializes the supplied **c_UInt64** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val The value to serialize.

- `ccArchive& operator|| (c_Int64& val);`

  Serializes the supplied **c_Int64** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val The value to serialize.

- `ccArchive& operator|| (unsigned int& val);`

  Serializes the supplied **unsigned int** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val The value to serialize. The value is always serialized as an unsigned 32-bit integer.

- `ccArchive& operator|| (int& val);`

  Serializes the supplied **int** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val — The value to serialize. The value is always serialized as a 32-bit integer.

- `ccArchive& operator|| (float& val);`

  Serializes the supplied **float** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val — The value to serialize. The value is serialized using the IEEE 754 format.

- `ccArchive& operator|| (double& val);`

  Serializes the supplied **double** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Parameters

  val — The value to serialize. The value is serialized using the IEEE 754 format.

- `ccArchive& operator|| (cmStd string& val);`

  Serializes the supplied **string** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

  Upon storing, the supplied **string** is converted to Unicode using the application's current C-runtime locale setting before being archived. Upon loading, if the data being read from the archive is Unicode, then the string is converted to an 8-bit representation using the application's current C-runtime locale setting.

  Parameters

  val — The value to serialize. The maximum length of a string that can be serialized using this operator is 32,767 bytes.

Throws

ccBadUnicodeChar

A character in *val* cannot be converted to Unicode using the current C-runtime locale setting (storing) or a Unicode character in the archive cannot be converted to an 8-bit representation using the current C-runtime locale setting (loading).

- `ccArchive& operator|| (cmStd wstring& val);`

Serializes the supplied **wstring** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

Parameters

val The value to serialize. The maximum length cmStd wstring that may be serialized is 32767 wchars.

- `ccArchive& operator|| (wchar_t& val);`

Serializes the supplied **wchar_t** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

Parameters

val The value to serialize.

- `ccArchive& operator|| (ccCvlString32& val);`

Serializes the supplied **ccCvlString32** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

Parameters

val The value to serialize.

- `ccArchive& operator|| (bool& val);`

Serializes the supplied **bool** value to or from this **ccArchive**. The value is serialized to the archive if the **ccArchive** is storing; the value is serialized from the **ccArchive** if the **ccArchive** is loading.

Parameters

val The value to serialize.

operator<<

```
ccArchive& operator<< (c_UInt8 val);
ccArchive& operator<< (c_Int8 val);
ccArchive& operator<< (char val);
ccArchive& operator<< (c_UInt16 val);
ccArchive& operator<< (c_Int16 val);
ccArchive& operator<< (c_UInt32 val);
ccArchive& operator<< (c_Int32 v) val);
ccArchive& operator<< (c_UInt64 val);
ccArchive& operator<< (c_Int64 val);
ccArchive& operator<< (unsigned int val);
ccArchive& operator<< (int val);
ccArchive& operator<< (float val);
ccArchive& operator<< (double val);
ccArchive& operator<< (cmStd string& val);
ccArchive& operator<< (const cmStd wstring& val);
ccArchive& operator<< (wchar_t& val);
ccArchive& operator<< (const ccCvlString32& val);
ccArchive& operator<< (bool& val);
```

Each of the overloads of the **operator<<** function is identical to the corresponding **operatorII** function except that the **ccArchive** must be storing.

operator>>

```
ccArchive& operator>> (c_UInt8& val);
ccArchive& operator>> (c_Int8& val);
ccArchive& operator>> (char& val);
ccArchive& operator>> (c_UInt16& val);
ccArchive& operator>> (c_Int16& val);
ccArchive& operator>> (c_UInt32& val);
ccArchive& operator>> (c_Int32& val);
ccArchive& operator>> (c_UInt64& val);
ccArchive& operator>> (c_Int64& val);
ccArchive& operator>> (unsigned int& val);
ccArchive& operator>> (int& val);
ccArchive& operator>> (float& val);
ccArchive& operator>> (double& val);
ccArchive& operator>> (cmStd string& val);
ccArchive& operator>> (cmStd wstring& val);
ccArchive& operator>> (wchar_t& val);
ccArchive& operator>> (ccCvlString32& val);
ccArchive& operator>> (bool& val);
```

Each of the overloads of the **operator>>** function is identical to the corresponding **operatorII** function except that the **ccArchive** must be loading.

Public Member Functions

raw

```
void raw(char* data, c_Int32 count);
```

Stores or loads raw data to or from the **ccArchive** without any format modification. The data is loaded or stored according to this **ccArchive**'s mode.

Parameters

data	The data to store or load. *data* must not be null.

count The number of bytes to store or load. *count* must not be less than 0.

Throws

ccArchive::Eof No more input data (loading only). The archive is left positioned at the end of the input stream.

ccArchive::BadBuffer
This **ccArchive** is broken and is unusable (loading or storing). If you are storing data, the disk may be full.

mode

```
Direction mode() const;

void mode(Direction mode);
```

- `Direction mode() const;`

  Returns a value indicating whether this **ccArchive** is loading or storing. This function returns one of the following values:

  ccArchive::eLoading
  ccArchive::eStoring

  Throws

  ccArchive::NotSeekable
  The ccArchive is not seekable.

- Sets this **ccArchive** to the specified mode (loading or storing). Cognex recommends that you do not call this function. Instead, specify the mode when you instantiate the **ccArchive**-derived concrete class.

  Parameters

  mode The mode to set. *mode* must be one of the following values:

  ccArchive::eLoading
  ccArchive::eStoring

  Notes

  In order to call this function, **ccArchive::isSeekable()** must return true. If **ccArchive::isReadOnly()** returns true, you cannot set this archive to storing.

  Throws

  ccArchive::NotSeekable
  The ccArchive is not seekable.

isLoading

```
bool isLoading() const;
```

Returns true if this **ccArchive** is loading, false otherwise.

isStoring

```
bool isStoring() const;
```

Returns true if this **ccArchive** is storing, false otherwise.

isReadOnly

```
bool isReadOnly() const;
```

Returns true if this **ccArchive** is read-only, false otherwise.

getByteOrder

```
Ordering getByteOrder() const;
```

Returns the endianness of this **ccArchive**. This function returns one of the following values:

ccArchive::eBigEndian
ccArchive::eLittleEndian

append

```
void append();
```

Sets this **ccArchive** to storing, seeks to the end, and prepares to accept additional stored objects. You must call **append()** when switching an archive constructed for loading to storing mode. Use this function to store objects to an archive that was constructed for loading.

Notes

This operation does not store anything into the **ccArchive**. In order to call this function, **ccArchive::isSeekable()** must return true. If **ccArchive::isReadOnly()** returns true, you cannot set this archive to storing.

Throws

ccArchive::NotSeekable
The ccArchive is not seekable.

sync

```
void sync();
```

Ensures that all of this **ccArchive**'s input or output data is flushed.

isSeekable

```
bool isSeekable() const;
```

Returns true if this **ccArchive** was constructed as a seekable archive.

version

```
c_Int32 version() const;
```

Returns the archive format version number of this archive.

Notes

It may not be possible to append to an archive with an old version number. Attempting an unsupported operation will throw *CannotWriteToOldArchive*.

tell

```
cmStd streampos tell() const;
```

Returns the **ccArchive**'s current offset from its beginning.

Throws

ccArchive::BadBuffer
The **ccArchive** is broken and is unusable.

---

seek

```
ccArchive& seek(cmStd streampos pos);

ccArchive& seek(cmStd streamoff offset,
  cmStd ios::seek_dir dir);
```

- `ccArchive& seek(cmStd streampos pos);`

Seeks to the specified location within this **ccArchive**. Cognex recommends that you do not use this function.

Parameters

pos The position to which to seek.

Throws

ccArchive::NotSeekable
The ccArchive is not seekable.

-
```
ccArchive& seek(cmStd streamoff offset,
  cmStd ios::seek_dir dir);
```

Seeks to the specified offset and direction from the current seek position within this **ccArchive**. Cognex recommends that you do not use this function.

Parameters

offset The offset from the current seek position.

dir The direction in which to seek.

Notes

You can only seek to an offset that lies within the extent of this **ccArchive**. A storing archive should be written sequentially. Previously written complex-persistent objects and pointers must not be overwritten.

Throws

ccArchive::BadBuffer
The **ccArchive** is broken and is unusable.

ccArchive::NotSeekable
The ccArchive is not seekable.

archiveAsData

```
ccArchive& archiveAsData (cmStd string&);
```

Archives the supplied string as 8-bit data. Unlike **operator ll()**, this function does not attempt to convert the supplied string to or from Unicode.

Parameters

string The data to archive.

Static Functions

archive8BitStringsAsLegacy

```
static void archive8BitStringsAsLegacy8BitData(
   bool useLegacy);

static bool archive8BitStringsAsLegacy8BitData();
```

- ```
  static void archive8BitStringsAsLegacy8BitData(
     bool useLegacy);
  ```

  Sets legacy archive mode for the current application. If *useLegacy* is true, all instances of **cmStd string** are archived as 8-bit data, for all instances of **ccArchive**. No attempt is made to interpret the bytes as characters using the current locale. Embedded NULLs are accepted and archived properly.

  Setting legacy mode to true overrides the normal behavior of **ccArchive::operatorll(cmStd string& val)**.

  Notice that the use of legacy mode is discouraged. Legacy behavior makes the interpretation of stored non-ASCII characters (outside of the range 0x00 to 0x7F) impossible unless you also know the locale in which they were encoded. The encoding of the bytes is not stored in the archive.
  ```

Parameters

useLegacy *true* to enable legacy mode, *false* to disable legacy mode.

- `static bool archive8BitStringsAsLegacy8BitData();`

  Returns *true* if legacy archive mode is set for the current application, false otherwise.

ccAutoSelectDefs

```
#include <ch_cvl/autoslct.h> // If not calling cfAutoSelect()
#include <ch_cvl/atslptmx.h> // If calling cfAutoSelect() with
                             // PatMax
#include <ch_cvl/atslcnls.h> // If calling cfAutoSelect() with
                             // CNLSearch

class ccAutoSelectDefs;
```

This class is a name space that holds enumerations and constants that are used with the Auto-select tool, and errors that the tool can throw.

cfAutoSelect<>() is a template function declared in *<ch_cvl/autoslct.h>*. As the definition does not provide any specialization for using this function with PatMax or CNLSearch, you must include either *<ch_cvl/atslptmx.h>* or *<ch_cvl/atslcnls.h>* to use this function with either of these tools.

ccAutoSelectDefs, **ccAutoSelectParams**, and **ccAutoSelectResult**, which are not templates, are also declared in *<ch_cvl/autoslct.h>*. If you want to use these classes in a compilation unit (for example, source code or header) that does not make a call to **cfAutoSelect()**, you can safely include only *<ch_cvl/autoslct.h>*. However, if you want to derive objects from these classes for use with PatMax or CNLSearch, you must include either *<ch_cvl/atslptmx.h>* or *<ch_cvl/atslcnls.h>*, respectively.

For example, if your program uses these source files:

- *a.cpp* --- which implements the base class, references defs, params, and results
- *b.cpp* --- which implements a derived class that runs Auto-select for PatMax
- *c.cpp* --- which implements a derived class that runs Auto-select for CNLSearch

then *a.cpp* can safely include *<ch_cvl/autoslct.h>*, but *b.cpp* must include *<ch_cvl/atslptmx.h>* for PatMax support, and *c.cpp* must include *<ch_cvl/atslcnls.cpp>* for CNLSearch support.

Enumerations

Direction

`enum Direction`

This enumeration defines the orientation of the axes about which model symmetry and orthogonality are measured.

Value	Meaning
eImageNormal	The axes are the image coordinate system x-axis and y-axis.
kDefaultDirection	The default direction (the default is *eImageNormal*).

ScoreCombineMethod

`enum ScoreCombineMethod`

This enumeration defines how the individual symmetry, orthogonality, and uniqueness scores are combined.

Value	Meaning
eGeometricMean	The geometric mean is computed.
eArithmeticMean	The arithmetic mean is computed.
kDefaultScoreCombineMethod	The default combination method (the default is *eGeometricMean*).

ccAutoSelectParams

```
#include <ch_cvl/autoslct.h> // If not calling cfAutoSelect()
#include <ch_cvl/atslptmx.h> // If calling cfAutoSelect() with
                             // PatMax
#include <ch_cvl/atslcnls.h> // If calling cfAutoSelect() with
                             // CNLSearch

class ccAutoSelectParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the run-time parameters for the Auto-select tool.

cfAutoSelect<>() is a template function declared in *<ch_cvl/autoslct.h>*. As the definition does not provide any specialization for using this function with PatMax or CNLSearch, you must include either *<ch_cvl/atslptmx.h>* or *<ch_cvl/atslcnls.h>* to use this function with either of these tools.

ccAutoSelectDefs, **ccAutoSelectParams**, and **ccAutoSelectResult**, which are not templates, are also declared in *<ch_cvl/autoslct.h>*. If you want to use these classes in a compilation unit (for example, source code or header) that does not make a call to **cfAutoSelect()**, you can safely include only *<ch_cvl/autoslct.h>*. However, if you want to derive objects from these classes for use with PatMax or CNLSearch, you must include either *<ch_cvl/atslptmx.h>* or *<ch_cvl/atslcnls.h>*, respectively.

For example, if your program uses these source files:

- *a.cpp* --- which implements the base class, references defs, params, and results
- *b.cpp* --- which implements a derived class that runs Auto-select for PatMax
- *c.cpp* --- which implements a derived class that runs Auto-select for CNLSearch

then *a.cpp* can safely include *<ch_cvl/autoslct.h>*, but *b.cpp* must include *<ch_cvl/atslptmx.h>* for PatMax support, and *c.cpp* must include *<ch_cvl/atslcnls.cpp>* for CNLSearch support.

Constructors/Destructors

ccAutoSelectParams

```
ccAutoSelectParams(bool enhancedMode = false);
```

Constructs a **ccAutoSelectParams** object that specifies whether the Auto-select tool should run in enhanced mode or non-enhanced mode. All other parameters are set to the following default values.

Parameter	Default Value
Direction	*ccAutoSelectDefs::kDefaultDirection* (currently *eImageNormal*)
Model size	64x64 pixels
Sampling rate	2
Score combination method	*ccAutoSelectDefs::kDefaultScoreCombineMethod* (currently *eGeometricMean*)
Score combination weights	Relative weight of symmetry score = 1 Relative weight of orthogonality score = 1 Relative weight of uniqueness score = 1
Maximum number of results	1
Window mask	Unbound 8-bit pel buffer
XY overlap	0.8

Parameters

enhancedMode If true, the Auto-select tool will run in enhanced mode; if false , the tool will run in non-enhanced mode (the default). There is no setter that will allow you to change this parameter later on.

Notes

Enhanced mode supports masking, overlap constraints, query operations on selected locations, and blanket search operations. When run in enhanced mode, the tool may return different results than in would be returned non-enhanced mode. You can enable enhanced mode only at the time the **ccAutoSelectParams** object is constructed.

Operators

operator==

```
bool operator==(const ccAutoSelectParams &rhs) const;
```

Compares two **ccAutoSelectParams** objects for equality. Returns true if all parameters in both objects are equal, false otherwise.

Parameters

rhs The **ccAutoSelectParams** object to be compared to this one.

Public Member Functions

direction

```
ccAutoSelectDefs::Direction direction() const;
```

```
void direction(ccAutoSelectDefs::Direction direction);
```

- `ccAutoSelectDefs::Direction direction() const;`

Returns the orientation of the axes about which pattern symmetry and orthogonality are computed. The returned direction is one of the following values:

ccAutoSelectDefs::eImageNormal

- `void direction(ccAutoSelectDefs::Direction direction);`

Sets the orientation of the axes about which pattern symmetry and orthogonality are computed.

Parameters

direction The axis orientation to set. Must be one of the following values:

ccAutoSelectDefs::eImageNormal
ccAutoSelectDefs::kDefaultDirection

Notes

In the current release, only the direction *ccAutoSelectDefs::eImageNormal* works.

modelSize

```
ccIPair modelSize() const;
```

```
void modelSize(const ccIPair &modelSize);
```

- `ccIPair modelSize() const;`

Returns the dimensions of the model training window in image coordinates.

- `void modelSize(const ccIPair &modelSize);`

  Sets the dimensions of the model training window in image coordinates.

  Parameters

  modelSize The model dimensions to set.

  Notes

  Invocations of **cfAutoSelect()** will throw *ccAutoSelectDefs::BadModel* if at least one of the components of the run parameter's model size either too small or too large to be able to obtain reliable data.

sample

```
c_Int16 sample() const;

void sample(c_Int16 sample);
```

- `c_Int16 sample() const;`

  Returns the sampling rate. The input image is sub-sampled by this factor before it is evaluated. The larger the sampling rate, the quicker the Auto-select tool will run, and the worse the quality of the result will be.

- `void sample(c_Int16 sample);`

  Sets the sampling rate. The input image is sub-sampled by this factor before it is evaluated. The larger the sampling rate, the quicker the Auto-select tool will run, and the worse the quality of the result will be.

  Parameters

  sample The sampling rate set. Do not specify a sampling rate larger than 8.

  Notes

  Invocations of **cfAutoSelect()** will throw *ccAutoSelectDefs::BadSample* if the sampling rate is less than or equal to 0.

scoreCombineMethod

```
ccAutoSelectDefs::ScoreCombineMethod scoreCombineMethod()
  const;

void scoreCombineMethod(
  ccAutoSelectDefs::ScoreCombineMethod method);
```

- `ccAutoSelectDefs::ScoreCombineMethod scoreCombineMethod() const;`

  Returns the score combination method configured for this **ccAutoSelectParams**. The returned value is one of the following values:

  ccAutoSelectDefs::eGeometricMean
  ccAutoSelectDefs::eArithmeticMean

- `void scoreCombineMethod( ccAutoSelectDefs::ScoreCombineMethod method);`

  Sets the score combination method for this **ccAutoSelectParams**. Specify the geometric mean if you only want a high overall score if all component scores are high; specify the arithmetic mean if you want a high overall score if *any* component score is high.

  Parameters

  method — The score combination method to set. Must be one of:

  ccAutoSelectDefs::eGeometricMean
  ccAutoSelectDefs::eArithmeticMean

scoreCombineWeights

```
cc3Vect scoreCombineWeights() const;

void scoreCombineWeights(const cc3Vect &weights);
```

- `cc3Vect scoreCombineWeights() const;`

  Returns a vector containing the weights to apply to the component scores when computing the overall score. The first element of the returned vector is the weight for the symmetry score, the second element is the weight for the orthogonality score, and the third element is the weight for the uniqueness score.

  The weights vector is normalized to a unit sum.

- `void scoreCombineWeights(const cc3Vect &weights);`

  Sets the weights to apply to the component scores when computing the overall score. Set the first element of *weights* to be the weight for the symmetry score, the second element to be the weight for the orthogonality score, and the third element to be the weight for the uniqueness score.

  scoreCombineWeights() normalizes the values you supply to a unit sum; only the relative values of the weights is considered.

  Parameters

  weights The weights to set. All of the elements of *weights* must be greater than or equal to 0, and at least one of the elements must be greater than 0.

  Notes

  Invocations of **cfAutoSelect()** will throw *ccAutoSelectDefs::BadWeights* if any of the run parameter object's weights is negative, or if all are 0.

maxNumResult

```
c_Int16 maxNumResult() const;

void maxNumResult(c_Int16 maxNumResult);
```

- `c_Int16 maxNumResult() const;`

  Returns the maximum number of results this **ccAutoSelectParams** is configured to return. The tool might return fewer than the specified number of results; it will never return more results than you specify.

- `void maxNumResult(c_Int16 maxNumResult);`

  Sets the maximum number of results that this **ccAutoSelectParams** returns. The tool will never return more results than you specify, but it may return fewer.

  Parameters

  maxNumResult The maximum number of results to return

  Notes

  Invocations of **cfAutoSelect()** will throw *ccAutoSelectDefs::BadResult* if *maxNumResult* is not positive.

  Only the 2***maxNumResult()** results with the best symmetry scores are used to compute overall scores. This means that if you set a low value for the maximum number of results, you might exclude results with low symmetry scores but high orthogonality and uniqueness scores.

windowMask

```
const ccPelBuffer_const<c_UInt8> &windowMask() const;

void windowMask(
  const ccPelBuffer_const<c_UInt8> &windowMask);
```

- `const ccPelBuffer_const<c_UInt8> &windowMask() const;`

  Retrieves the window mask, an 8-bit pel buffer that is the size of the model window.

  Throws

  ccAutoSelectDefs::NotEnhancedMode
  Enhanced mode is not enabled.

- `void windowMask(`
  `  const ccPelBuffer_const<c_UInt8> &windowMask);`

  Sets the window mask.

  Parameters

  windowMask An 8-bit pel buffer. The window mask should either be unbound or have the same size as the model window, as returned by **ccAutoSelectParams::modelSize()**.

  Notes

  Invocations of **cfAutoSelect()** will throw *ccAutoSelectDefs::BadMask* if the window mask is a bound pel buffer of any size other the model size.

  Throws

  ccAutoSelectDefs::NotEnhancedMode
  Enhanced mode is not enabled.

xyOverlap

```
double xyOverlap() const;

void xyOverlap(double xyOverlap);
```

- `double xyOverlap() const;`

  Retrieves the XY overlap value. The overlap value is approximately the ratio of the overlapping area to the total area of the pattern. Two pattern instances are considered to overlap if their overlap value is greater than the XY overlap, and only one of them will be returned as a result from a search operation.

  Notes

  Meaningful XY overlap values range from 0.0 to 1.0. A value of 0.0 specifies that no overlap is allowed. A value of 1.0 specifies that any amount of overlap is allowed (overlap checking is disabled).

Throws

ccAutoSelectDefs::NotEnhancedMode
Enhanced mode is not enabled.

- `void xyOverlap(double xyOverlap);`

Sets the XY overlap value that determines whether two pattern instances are considered the same result, and only one of them will be returned from a search operation.

Parameters

xyOverlap The XY overlap value to set. Must be in the range of 0.0 to 1.0.

Value	Meaning
0.0	No overlap is allowed.
1.0	Any amount of overlap is allowed (overlap checking is disabled).

Throws

ccAutoSelectDefs::NotEnhancedMode
Enhanced mode is not enabled.

ccAutoSelectDefs::BadParams
xyOverlap is outside the range of 0.0 through 1.0, inclusive.

isEnhancedMode

```
bool isEnhancedMode() const;
```

Returns true if the enhanced mode is enabled, false otherwise.

Notes

Enhanced mode can be enabled only at the time the **ccAutoSelectParams** object is constructed.

ccAutoSelectResult

```
#include <ch_cvl/autoslct.h> // If not calling cfAutoSelect()
#include <ch_cvl/atslptmx.h> // If calling cfAutoSelect() with
                             // PatMax
#include <ch_cvl/atslcnls.h> // If calling cfAutoSelect() with
                             // CNLSearch

class ccAutoSelectResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the result of running the Auto-select tool on the input image. The global function **cfAutoSelect()** returns a vector of **ccAutoSelectResult** objects.

cfAutoSelect<>() is a template function declared in *<ch_cvl/autoslct.h>*. As the definition does not provide any specialization for using this function with PatMax or CNLSearch, you must include either *<ch_cvl/atslptmx.h>* or *<ch_cvl/atslcnls.h>* to use this function with either of these tools.

ccAutoSelectDefs, **ccAutoSelectParams**, and **ccAutoSelectResult**, which are not templates, are also declared in *<ch_cvl/autoslct.h>*. If you want to use these classes in a compilation unit (for example, source code or header) that does not make a call to **cfAutoSelect()**, you can safely include only *<ch_cvl/autoslct.h>*. However, if you want to derive objects from these classes for use with PatMax or CNLSearch, you must include either *<ch_cvl/atslptmx.h>* or *<ch_cvl/atslcnls.h>*, respectively.

For example, if your program uses these source files:

- *a.cpp* --- which implements the base class, references defs, params, and results
- *b.cpp* --- which implements a derived class that runs Auto-select for PatMax
- *c.cpp* --- which implements a derived class that runs Auto-select for CNLSearch

then *a.cpp* can safely include *<ch_cvl/autoslct.h>*, but *b.cpp* must include *<ch_cvl/atslptmx.h>* for PatMax support, and *c.cpp* must include *<ch_cvl/atslcnls.cpp>* for CNLSearch support.

Constructors/Destructors

ccAutoSelectResult

```
ccAutoSelectResult();
```

Constructs a default **ccAutoSelectResult** object with its location set to (0,0) and all scores set to 0.0.

Notes

You should not attempt to create **ccAutoSelectResult** objects yourself.

Public Member Functions

location

```
ccIPair location() const;
```

Returns the upper-left corner of the returned model location window for this **ccAutoSelectResult**, in image coordinates.

score

```
double score() const;
```

Returns the overall score for this **ccAutoSelectResult**. The overall score is computed from the three component scores according to the weighting and combination method parameters you specified.

symmetryScore

```
double symmetryScore() const;
```

The symmetry score for this **ccAutoSelectResult**. This score is a whether or not this window is more symmetrical than nearby locations in the image.

orthoScore

```
double orthoScore() const;
```

The orthogonality score for this **ccAutoSelectResult**. This score is a measure of the degree to which the model window contains a balance of strong horizontal and vertical features. A low orthogonality score can indicate few horizontal and vertical features or an imbalance between horizontal and vertical features.

uniqueScore

```
double uniqueScore() const;
```

The uniqueness score for this **ccAutoSelectResult**. The uniqueness score is a measure of how unique the model window is when compared to the rest of the input image, using the pattern-location tool run-time parameters you specified when you called **cfAutoSelect()**.

ccBallPatternAlignBallResult

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignBallResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class holds the result of ball inspection. Instances of this class are created only by the tool. The **ccBallPatternAlignBallResult** class has const getters for all of the available data.

Constructors/Destructors

ccBallPatternAlignBallResult

```
ccBallPatternAlignBallResult();
```

Default constructor.

Notes

Compiler generated copy constructor, assignment operator, and destructor are used.

Operators

operator==

```
bool operator==(const ccBallPatternAlignBallResult &rhs)
const;
```

Equality test operator. Returns true if this object is exactly equal to *rhs*. Returns false otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

found

```
bool found() const;
```

Returns true if the ball is found; otherwise, returns false.

ccBallPatternAlignBallResult

position

```
const cc2Vect & position() const;
```

If **found()** returned true, returns the found ball position in client space.

size

```
double size() const;
```

If **found()** returned true, returns the found ball size in physical units.

ccBallPatternAlignDefs

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignDefs;
```

A name space that holds enumerations used with the Ball Pattern Align tool classes.

Enumerations

DOF

```
enum DOF;
```

Degrees of freedom.

Value	Meaning
eAngle =1	Whether rotation is enabled.
eUniformScale = 2	Whether uniform scale is enabled.
eXScale = 4	Whether X scale is enabled.
eYScale = 8	Whether Y scale is enabled.

Polarity

```
enum Polarity;
```

Supported polarities.

Value	Meaning
eLightOnDark = 0x0001	Light on dark polarity.
eDarkOnLight = 0x0002	Dark on light polarity.
eUnknown = *eDarkOnLight* | *eLightOnDark*	Unknown polarity.
kDefaultPolarity = *eLightOnDark*	The default is light on dark.

ccBallPatternAlignDefs

RefineMode `enum RefineMode;`

The refine mode.

Value	Meaning
eRefineByImageFeatures = 1	Refine by image features.
eRefineByTrainedBlobs = 2	Refine by trained blobs.
kRefineDefault = *eRefineByImageFeatures*	The default is refining by image features.

DrawMode `enum DrawMode;`

The drawing mode.

Value	Meaning
eDrawOrigin = 0x001	Whether to draw the origin.
eDrawLabel = 0x002	Whether to draw the label.
eDrawBoundingBox = 0x004	Whether to draw the bounding box.
eDrawBlobs = 0x008	Whether to draw the blobs.
eDrawEdgelets = 0x010	Whether to draw the edgelets.
eDrawCircles = 0x020	Whether to draw the circles.
eDrawStandard = *eDrawOrigin* | *eDrawBoundingBox* | *eDrawEdgelets*	The standard is drawing the origin, the bounding box, and the edgelets.

ccBallPatternAlignModel

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignModel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class is the central object for the Ball Pattern Align tool. It contains the trained representation of the model, and the functions to do the training. This class also features a function that runs the model on an image and produces results.

Constructors/Destructors

ccBallPatternAlignModel

```
ccBallPatternAlignModel();

~ccBallPatternAlignModel();

ccBallPatternAlignModel(
  const ccBallPatternAlignModel& rhs);
```

- `ccBallPatternAlignModel();`

  Construct an untrained alignment model with default values.

- `~ccBallPatternAlignModel();`

  Destructor.

- ```
  ccBallPatternAlignModel(
    const ccBallPatternAlignModel& rhs);
  ```

  Copy constructor.

  Parameters

rhs	The source of the copy.
  ```

Operators

operator=

```
const ccBallPatternAlignModel& operator=(
  const ccBallPatternAlignModel& rhs);
```

Assignment operator.

Parameters

rhs The object to assign.

Public Member Functions

origin

```
cc2Vect origin() const;

void origin(const cc2Vect &origin);
```

- `cc2Vect origin() const;`

Returns the model's origin.

- `void origin(const cc2Vect &origin);`

Sets the model's origin.

Parameters

origin The origin to be set.

Notes

The origin is the translation component of the model coordinate system (see above). The origin is specified in the client coordinate system of the training image. The origin is a point attached to the model, but it may reside outside of the model. When an instance of the model is found, the result location is defined by the position of the origin in the search image.

The location of the origin in the image is trained as the origin. Found poses will transform this origin into the runtime image.

Setting **trainClientFromModel()** changes the origin and setting the origin changes the translation component of **trainClientFromModel()**, but does not affect the matrix portion of the transform.

The origin can be read or written for either a trained or an untrained model. Changing the origin does not require that the model be retrained.

Setting the origin affects not only the **location()** of the result, but also the **pose()**.

The default is cc2Vect(0, 0)

trainClientFromModel

```
cc2Xform trainClientFromModel();

void trainClientFromModel(const cc2Xform&);
```

- `cc2Xform trainClientFromModel();`

  Returns the mapping from model coordinates to training image client coordinates.

- `void trainClientFromModel(const cc2Xform&);`

  Sets the mapping from model coordinates to training image client coordinates.

  Notes

  Model coordinates are a client-controlled coordinate system that specifies how to match the model to an image and report result poses.
  A result pose is a mapping from model coordinates to run-time client coordinates, and likewise the generalized DOFs define the search range as a mapping from model coordinates to run-time client coordinates. Model coordinates may be set in all 6 DOFs. Note that model coordinates includes the model's origin as its translation component, and that setting the model coordinates with this member always sets the origin. Model coordinates are never modified by the act of training; they are relative to the client coordinates established by the most recent training, if any.

  trainClientFromModel() cannot be used after training, although the translation component can be changed using **origin()**

  Throws

  AlreadyTrained The model is already trained.

  BadParams The new xform is singular.

  The default is identity **cc2Xform**.

train

```
void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccBallPatternAlignTrainParams& trainParams,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccBallPatternAlignTrainParams &trainParams,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
  const cmStd vector<ccCircle>& blobs,
  const ccBallPatternAlignTrainParams& trainParams,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const cmStd vector<ccCircle>& blobs,
  const ccBallPatternAlignTrainParams& trainParams,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);
```

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const ccBallPatternAlignTrainParams& trainParams,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0);
  ```

  Image-based training. Extracts the grey-scale blobs from the given image, and trains this model from the extracted blobs.

  Parameters

image	The image.
trainParams	The training parameters.
diagobj	An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.
diagFlags	Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values:
  ```

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

InsufficientFeatures
Not enough blobs are found for successful training.

BadImage
The image is unbound or has a Null window (that is, height==0 OR width==0)

BadImageSize
The image is too small.

Notes

If this model is already trained, it is automatically untrained and retrained.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccBallPatternAlignTrainParams &trainParams,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0);
  ```

Image-based training. Extracts the grey-scale blobs from the given image and mask, and trains this model from the extracted blobs.

Parameters

image The image.

mask The mask.

trainParams The training parameters.

diagobj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:
  ```

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

InsufficientFeatures
Not enough blobs are found for successful training.

BadImage The image is or unbound or has a Null window (that is, height==0 OR width==0)
OR
the mask is unbound
OR
the mask does not have the same window as the image
OR
any of the mask pels are values which are not 0 or 255

BadImageSize The image is too small.

Notes

If this model is already trained, it is automatically untrained and retrained.

In the mask, pixels whose grey level is 0 indicate “don't care” pixels in the training image. Mask pixels whose grey level is 255 denote “care” pixels in the training image.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const cmStd vector<ccCircle>& blobs,
    const ccBallPatternAlignTrainParams& trainParams,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0);
  ```

Geometric training. Trains this model from the given circular blobs, together with the given image.

Parameters

image The image.

blobs The blobs.

trainParams The training parameters.

diagobj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.
  ```

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

InsufficientFeatures
Not enough blobs are supplied for successful training.

BadImage The image is unbound or has a Null window (that is, height==0 OR width==0).

BadImageSize The image is too small.

Notes

This training is the same as the image-based training, except that it does not extract the grey-scale blobs from the image; instead, it uses the user-provided circular blobs. It is useful in the case that the user has obtained the blobs elsewhere before training this model. Note that both the input image and user-provided blobs are used in training; it is required that the user-provided blobs match with the actual blobs in the image.

If this model is already trained, it is automatically untrained and retrained.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
   const ccPelBuffer_const<c_UInt8>& mask,
   const cmStd vector<ccCircle>& blobs,
   const ccBallPatternAlignTrainParams& trainParams,
   ccDiagObject* diagobj=0,
   c_UInt32 diagFlags=0);
  ```

Geometric training. Trains this model from the given circular blobs, together with the given image and mask.

Parameters

image The image.

mask The mask.
  ```

blobs The blobs.

trainParams The training parameters.

diagobj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

InsufficientFeatures
Not enough blobs are supplied for successful training.

BadImage The image is unbound or has a Null window (that is, height==0 OR width==0)
OR
the mask is unbound
OR
the mask does not have the same window as the image
OR
any of the mask pels are values which are not 0 or 255

BadImageSize The image is too small.

Notes

This training is the same as the image-based training, except that it does not extract the grey-scale blobs from the image; instead, it uses the user-provided circular blobs. It is useful in the case that the user has obtained the blobs elsewhere before training this model. Note that both the input image and user-provided blobs are used in training; it is required that the user-provided blobs match with the actual blobs in the image.

If this model is already trained, it is automatically untrained and retrained.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

untrain

```
void untrain();
```

Untrains this model. Releases any saved training data. It has no effect if this model is currently not trained.

trainParams

```
const ccBallPatternAlignTrainParams& trainParams() const;
```

Returns the *trainParams* used to train this model.

Throws

NotTrained This model is not trained.

trainImage

```
ccPelBuffer_const<c_UInt8> trainImage() const;
```

Returns the *image* used to train this model.

Throws

NotTrained This model is not trained.

trainMask

```
ccPelBuffer_const<c_UInt8> trainMask() const;
```

Returns the *mask* image used when this model was trained.

Throws

NotTrained This model is not trained.

Notes

Returns an unbound pel buffer if the model was trained without a mask.

isGeometricTraining

```
bool isGeometricTraining() const;
```

Returns whether a Geometric Training was performed.

Throws

NotTrained This model is not trained.

trainedBlobs

```
std::vector<ccCircle> trainedBlobs() const;
```

Returns the trained blobs if image-based training was performed; returns the user-supplied blobs if the geometric training was performed. The blobs are returned in client coordinates.

Throws

NotTrained This model is not trained.

Notes

It is usually fine if image-based training did not return all possible blobs in the image. Not all but enough blobs need to be found for successful training.

trainedBlobCount

```
c_Int32 trainedBlobCount() const;
```

Returns the count of the blobs returned by **trainedBlobs()**.

Throws

NotTrained This model is not trained.

trainedBlobMeanDiameter

```
double trainedBlobMeanDiameter() const;
```

Returns the mean diameter, in client units, of the blobs returned by **trainedBlobs()**.

Throws

NotTrained This model is not trained.

Notes

Diameters of trained blobs by image-based training should not be considered a high-accuracy measurement.

trainedPatternBoundingBox

```
ccAffineRectangle trainedPatternBoundingBox() const;
```

Returns client-coordinates-aligned bounding box, in client coordinates, of the blobs returned by **trainedBlobs()**.

Throws

NotTrained This model is not trained.

Notes

This includes sufficient padding to locate the model using this trained tool.

trainedPolarity

```
ccBallPatternAlignDefs::Polarity trainedPolarity() const;
```

Returns polarity chosen by image-based training.

Throws

NotTrained This model is not trained or Geometric Training was performed.

Notes

This will always be a unique polarity. Training can choose a polarity, but never trains multiple polarities.

trainedCoarseAlignGrainLimit

```
double trainedCoarseAlignGrainLimit() const;
```

Returns the trained coarse grain limit for alignment.

Throws

NotTrained This model is not trained.

trainedFineAlignGrainLimit

```
double trainedFineAlignGrainLimit() const;
```

Returns the trained fine grain limit for alignment.

Throws

NotTrained This model is not trained.

trainedRefineGrainLimit

```
double trainedRefineGrainLimit() const;
```

Returns the trained refine grain limit.

Throws

NotTrained This model is not trained.

isTrained

```
bool isTrained() const;
```

Returns true if model is currently trained, false otherwise.

---

run

```
void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccBallPatternAlignRunParams &runParams,
  ccBallPatternAlignResultSet &resultSet,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0) const;

void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccBallPatternAlignRunParams &runParams,
```

```
  ccBallPatternAlignResultSet &resultSet,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0) const;
```

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccBallPatternAlignRunParams &runParams,
    ccBallPatternAlignResultSet &resultSet,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0) const;
  ```

  Run this model on the given image with the given runtime parameters. The supplied *resultSet* will be cleared and then filled with new results, in order of decreasing score.

  Parameters

image	The image.
runParams	The runtime parameters.
resultSet	The result set.
diagobj	An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.
diagFlags	Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

  Notes

  This function respects CVL timeouts. If a timeout occurs, no valid results are produced by the tool.

  This **run()** function can be called for models which were trained using masks, as well as by models that were trained without masks.

  Throws

NotTrained	This model is currently not trained.
BadImage	The *inputImage* is unbound.
  ```

BadParams One of the dof ranges in the *runParams* (composed by the *startPose*) exceeds the corresponding dof range in the *trainParams*.

ccTimeout::Expired
This function executes longer than the timeout specified by the user. If this occurs, no valid results are produced by the tool.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccBallPatternAlignRunParams &runParams,
    ccBallPatternAlignResultSet &resultSet,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0) const;
  ```

  Run this model on the given image with the given runtime parameters. The supplied *resultSet* will be cleared and then filled with new results, in order of decreasing score.

  Parameters

  image The image.

  mask The mask.

  runParams The runtime parameters.

  resultSet The result set.

  diagobj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff
  ```

Notes

This function respects CVL timeouts. If a timeout occurs, no valid results are produced by the tool.

This **run()** function can be called for models which were trained using masks, as well as by models which were trained without masks.

Throws

NotTrained	This model is currently not trained.
BadImage	The image is unbound OR the mask is unbound OR the mask does not have the same window as the image OR any of the mask pels are values which are not 0 or 255.
BadParams	One of the dof ranges in the *runParams* (composed by the *startPose*) exceeds the corresponding dof range in the *trainParams*.
ccTimeout::Expired	This function executes longer than the timeout specified by the user. If this occurs, no valid results are produced by the tool.

drawFeatures

```
void drawFeatures(ccGraphicList& graphicList,
 const cc2XformLinear& pose,
 const ccColor& color) const;
```

Draws the trained blobs into *graphicList*.

Parameters

graphicList	The graphic list.
pose	The pose.
color	The color.

drawAlignFeatures

```
void drawAlignFeatures(ccGraphicList& graphicList,
 const cc2XformLinear& pose,
 const ccColor& color) const;
```

Draws the edgelets used for refinement into *graphicList*.

Parameters

graphicList	The graphic list.

pose	The pose.
color	The color.

ccBallPatternAlignModel

ccBallPatternAlignResult

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single Ball Pattern Align tool result. Instances of this class are created only by the tool. The **ccBallPatternAlignResult** class has const getters for all of the available data.

Constructors/Destructors

ccBallPatternAlignResult

```
ccBallPatternAlignResult();
```

Constructs this object with no useful data.

Notes

Compiler generated copy constructor, assignment operator, and destructor are used.

Operators

operator==

```
bool operator==(const ccBallPatternAlignResult &rhs) const;
```

Equality test operator. Returns true if this object is exactly equal to *rhs*. Returns false otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

location

```
cc2Vect location() const;
```

Returns the location of the model origin in the client coordinate system of the runtime image.

The default is (0,0).

pose

```
const cc2Xform& pose() const;
```

Returns a transform from the training time client coordinates (translated to the model origin) to runtime client coordinates. Map **cc2Vect(**0,0**)** through this to get the location of the model origin in runtime client coordinates.

The default is *cc2Xform::I*

angle

```
ccDegree angle() const;
```

Returns the angle of the located model in the client coordinate system of the runtime image.

The default is 0.

xScale

```
double xScale() const;
```

Returns the X scale of the located model.

yScale

```
double yScale() const;
```

Returns the Y scale of the located model.

scale

```
double scale() const;
```

Returns the scale of the located model.

The default is 1.

score

```
double score() const;
```

Returns the score, a number between 0.0 and 1.0.

The default is 0.

accepted

```
bool accepted() const;
```

Returns true if the score was >= the runtime parameter accept threshold, otherwise returns false.

The default is false.

matchRegion

```
const ccAffineRectangle& matchRegion() const;
```

Returns an affine rectangle which describes the location of the matched model in the runtime image's client coordinates.

The default is **ccAffineRectangle()**

balls

```
const cmStd vector<ccBallPatternAlignBallResult> & balls()
const;
```

Return a vector of results for balls that are specified by **ccBallPatternAlignModel::trainedBlobs()**.

Notes

Only balls that are specified by **ccBallPatternAlignModel::trainedBlobs()** are included in the returned vector.

draw

```
void draw(ccGraphicList& graphicList,
  c_UInt32 drawMode =
  ccBallPatternAlignDefs::eDrawStandard,
  const ccCvlString& label = ccCvlString(),
  const ccBallPatternAlignModel* model = 0) const;
```

Appends graphics to *graphicList* in client coordinate system for this result.

Parameters

graphicList	The graphic list.
drawMode	The draw mode.
label	The label.
model	The model.

drawMode is used to select which graphics are drawn and must be a bitwise combination of the following flags (defined in **ccBallPatternAlignDefs**):

eDrawOrigin: Draw a cross at this result's location. The cross is rotated as the angle of this result.

eDrawLabel: Label the origin of this result with the provided label string.

eDrawBoundingBox: Draw a bounding box around this result representing matched region.

eDrawBlobs: Draw the trained blobs. Ignored if *model* == 0. Graphics will only be meaningful if model points to the model that generated this result and it has not been since modified.

eDrawEdgelets: Draw the edgelets used for refinement. The colors of the edgelets are dependent on whether *saveMatchInfo* was set in runtime params when running the tool to generate this result.

- If *saveMatchInfo* was set, the colors of the edgelets show the match info.
- If *saveMatchInfo* was not set, the edgelets are shown in a single color (either green or red, depending on whether this result was accepted). Additionally, if saveMatchInfo was not set, graphics will only be meaningful if model points to the model that generated this result and it has not been since modified.

eDrawCircles: Draw the balls. Draw found balls in green and not-found balls in red.

If **accepted()** is true for this result, all graphics, except *eDrawEdgelets*, are drawn in green; otherwise, they are drawn in red.

Notes

model is required for *eDrawBlobs* and for *eDrawEdgelets*; otherwise, it can be 0.

ccBallPatternAlignResultSet

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains all of the results from a single invocation of the **ccBallPatternAlignModel::run()** function.

Constructors/Destructors

ccBallPatternAlignResultSet

```
ccBallPatternAlignResultSet();
```

Creates a result set with no results.

Notes

Compiler generated copy constructor, assignment operator, and destructor are used.

Operators

operator==

```
bool operator==(const ccBallPatternAlignResultSet &rhs)
const;
```

Equality test operator. Returns true if this object is exactly equal to *rhs*. Returns false otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

numFound

```
c_Int32 numFound() const;
```

Returns the number of results in this result set.

ccBallPatternAlignResultSet

results

```
const cmStd vector<ccBallPatternAlignResult>& results()
const;
```

Returns all results. The returned vector is of size **numFound()**.

ccBallPatternAlignRunParams

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class holds the runtime parameters used by the Ball Pattern Align tool. The parameters are defaulted at construction, and may be read/written via getters/setters.

Constructors/Destructors

ccBallPatternAlignRunParams

```
ccBallPatternAlignRunParams();
```

Constructs a ccBallPatternAlignRunParams object using the default values.

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Operators

operator==

```
bool operator==(
  const ccBallPatternAlignRunParams &rhs) const;
```

Equality test operator. Returns true if this object is exactly equal to *rhs*. Returns false otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

acceptThreshold

```
double acceptThreshold() const;
void acceptThreshold(double a);
```

- `double acceptThreshold() const;`

  Returns the threshold on the score of results.

- `void acceptThreshold(double a);`

  Sets the threshold on the score of results.

  This is used to set the “accepted” field of each result as follows:

  accepted = (score >= acceptThreshold);

  Parameters

  a The threshold.

  Throws

  BadParams

  A threshold value is supplied that is less than zero or greater than one.

  The default is 0.5.

numToFind

```
c_Int32 numToFind() const;
void numToFind(c_Int32 n);
```

- `c_Int32 numToFind() const;`

  Returns the number of results to look for.

- `void numToFind(c_Int32 n);`

  Sets the number of results to look for.

  The **ccBallPatternAlignModel::run()** functions may not return exactly **numToFind()** results. Fewer results will be returned if less than *numToFind* were found.

Parameters

n The number of results to look for.

Throws

BadParams

n <= 0

The default is 1.

startPose

```
cc2XformLinear startPose() const;
void startPose(const cc2XformLinear &startPose);
```

- `cc2XformLinear startPose() const;`

  Returns the start pose.

- `void startPose(const cc2XformLinear &startPose);`

  Sets the start pose.

  Parameters

  startPose The start pose.

  Throws

  BadParams

  The given start pose is singular.

  Notes

  This start pose maps the model from training-time coordinates into run-time coordinates. If the start pose is exactly correct, then the found result will have exactly the same pose.

  The default is **cc2XformLinear()**.

saveMatchInfo

```
bool saveMatchInfo() const;
void saveMatchInfo (bool x);
```

- `bool saveMatchInfo() const;`

  Returns the flag indicating whether to save information necessary to generate a match display.

- `void saveMatchInfo (bool x);`

  Sets the flag indicating whether to save information necessary to generate a match display.

  Parameters

  x The flag.

  Notes

  If this flag is set, some extra time and extra memory is used for each result returned.

  The default is false.

translationUncertainty

```
ccRect translationUncertainty() const;

void translationUncertainty(
  const ccRect& translationUncertainty);
```

- `ccRect translationUncertainty() const;`

  Returns the translation uncertainty.

- `void translationUncertainty(`
  `const ccRect& translationUncertainty);`

  Sets the translation uncertainty.

  Parameters

  translationUncertainty

  The translation uncertainty.

  Throws

  ccBallPatternAlignDefs::BadParams

  A rect is given that does not contain the point (0, 0) and is not the rect (0, 0, 0, 0)
  or
  a rect is given that has HUGE_VAL or -HUGE_VAL for any corner or size except for the fully infinite rect (-HUGE_VAL,-HUGE_VAL,HUGE_VAL,HUGE_VAL).

  Notes

  The translation uncertainty is in client coordinate units and is aligned to the client coordinate axes. The rect may be (0, 0, 0, 0) to indicate a point search. Otherwise, the rect must contain the point (0, 0).

The default is **ccRect(**-*HUGE_VAL*,-*HUGE_VAL*,*HUGE_VAL*,*HUGE_VAL***)**;

xyOverlap

```
double xyOverlap() const;
void xyOverlap(double overlapThresh);
```

- `double xyOverlap() const;`

Returns the required xy overlap to consider two instances to be the same result.

- `void xyOverlap(double overlapThresh);`

Sets the required xy overlap to consider two instances to be the same result.

Two results are considered overlapping if the Overlap value is > *overlapThresh*.

Parameters

overlapThresh The overlap threshold.

Throws

BadParams

The argument is outside the range 0.0 to 1.0.

Notes

Overlap is (approximately) the ratio of overlap area to total area of the model.

Meaningful values range from 0.0 to 1.0.

overlapThresh	Effect
0	Results with non-null intersection overlap
1	No results overlap (overlap disabled)

The default is 0.8.

zoneEnable

```
c_UInt32 zoneEnable() const;
void zoneEnable(c_UInt32 enable);
```

- `c_UInt32 zoneEnable() const;`

Returns the zone enable for each degree of freedom. Non-zero bits in *enable* indicate degrees of freedom to be searched.

- `void zoneEnable(c_UInt32 enable);`

  Sets the zone enable for each degree of freedom. Non-zero bits in *enable* indicate degrees of freedom to be searched.

  If *zoneEnable* is set for a given DOF, **ccBallPatternAlignModel::run()** will search between zoneLow and zoneHigh. Otherwise, **ccBallPatternAlignModel::run()** will use the start pose value.

  Parameters

  enable The zone enable for each degree of freedom.

  Throws

  BadParams

  The given *enable* value is not:
  1) zero or
  2) created by a bitwise-OR of the values from *ccBallPatternAlignDefs::DOF*.

  Notes

  If *zoneEnable* is set for a given DOF, results produced by **ccBallPatternAlignModel::run()** may fall outside zoneLow and zoneHigh. If the DOF is not enalbed, results produced by **ccBallPatternAlignModel::run()** will always be at exactly the start pose for that DOF.

  The runtime zones are relative to the start pose. For example, if the start pose is rotated by 30 degrees, then enabling the rotation DOF by -10 degrees to +10 degrees would mean that the tool would search from +20 degrees to +40 degrees.

  The default is 0 (no DOFs are enabled).

zoneLow `double zoneLow (ccBallPatternAlignDefs::DOF dof) const;`

Returns the low zone parameter for the given DOF.

Parameters

dof The given DOF.

Throws

BadParams

The given DOF value is illegal. See the following table regarding what is legal.

For the defaults, see the table hereinafter.

zoneHigh `double zoneHigh (ccBallPatternAlignDefs::DOF dof) const;`

Returns the high zone parameter for the given DOF.

Parameters

dof The given DOF.

Throws

BadParams

The given DOF value is illegal. See the following table regarding what is legal.

For the defaults, see the table hereinafter.

zone

```
void  zone(ccBallPatternAlignDefs::DOF dof, double low,
  double high);
```

Sets the low and high zone parameters for the given DOF.

Parameters

dof The given DOF.

low The low zone.

high The high zone.

Throws

BadParams

Illegal low zone or high zone values are given for the given DOF. See the following table regarding what is legal.

Notes

The span of each DOF's runtime zone must be smaller than or equal to the span of the associated DOF's training zone; otherwise, a runtime exception will be thrown.

The runtime zones are relative to the start pose. For example, if the start pose is rotation by 30 degrees, then enabling the rotation DOF by -10 degrees to +10 degrees would mean that the tool would search from +20 degrees to +40 degrees.

For the defaults, see the table hereinafter.

DOF Summary Table

This table summarizes the degrees of freedom values for this tool.

DOF	Unit	Default low	Default high	Range	Restrictions
Angle	degrees	-45	+45	any	none
Uniform scale	multiplier	0.8	1.2	> 0	zoneLow is less than or equal to zoneHigh

DOF	Unit	Default low	Default high	Range	Restrictions
X scale	multiplier	0.8	1.2	> 0	zoneLow is less than or equal to zoneHigh
Y scale	multiplier	0.8	1.2	> 0	zoneLow is less than or equal to zoneHigh

Notes

- Uniform scale is equal scale in both the x and y dimensions.
- X scale and Y scale only scale in the given dimension.
- Scale is specified as a multiplier of the trained model size. For example, a uniform scale of 1.25 implies that the runtime (client coordinate) size of the model will be 25% larger in that dimension than the trained model.
- The angle zone is always searched from the zoneLow to the zoneHigh. So, setting zoneLow to 0 and zoneHigh to 180 searches one half of the 360 degree angle range – setting zoneHigh to 0 and zoneLow to 180 searches the other half.

zoneOverlap

```
double zoneOverlap(ccBallPatternAlignDefs::DOF dof) const;

void zoneOverlap(ccBallPatternAlignDefs::DOF dof,
  double overlapThresh);
```

- `double zoneOverlap(ccBallPatternAlignDefs::DOF dof) const;`

Returns the zone overlap parameter for the given DOF. Angle and Scale overlap values are specified in different units.

Parameters

dof The DOF.

- `void zoneOverlap(ccBallPatternAlignDefs::DOF dof, double overlapThresh);`

  Sets the zone overlap parameter for the given DOF. Angle and Scale overlap values are specified in different units.

  For *ccBallPatternAlignDefs::kAngle*, two results are considered overlapping if the absolute difference between their angles is <= *overlapThresh*. Therefore, if dof is *kAngle*, *overlapThresh* must be >= 0.

  For any scale DOF, *overlapThresh* is a ratio which is compared to the ratio of the result with greater scale to the one with smaller scale. If the ratio of the two results is <= the *overlapThresh* ratio, the results are considered overlapping. Since the ratio is compared to the greater scale over the smaller scale, the *overlapThresh* must be >= 1.0. For example, if dof is one of the scale degrees of freedom, and *overlapThresh* = 1.2, then two results, one with a scale of 1.25 and one with a scale of 2.5, would not overlap in scale since 2.5/1.25 is greater than 1.2.

  Parameters

dof	The DOF.
overlapThresh	The zone overlap parameter to be set for the given DOF.

  Throws

BadParams	An illegal *dof* value is supplied to this setter or the *dof* value is out of range as specified above.

  The default is:

  Angle: 360.0.
  Uniform, X, and Y scales: 1.4.

ccBallPatternAlignRunParams

ccBallPatternAlignTrainParams

```
#include <ch_cvl/bpalign.h>

class ccBallPatternAlignTrainParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class holds the traintime parameters used by the Ball Pattern Align tool. The parameters are defaulted at construction, and may be read/written via getters/setters.

Constructors/Destructors

ccBallPatternAlignTrainParams

```
ccBallPatternAlignTrainParams();
```

Constructs the object using the default values.

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Operators

operator==

```
bool operator==(
  const ccBallPatternAlignTrainParams &rhs) const;
```

Equality test operator. Returns true if this object is exactly equal to *rhs*. Returns false otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

diameterRange

```
ccRange diameterRange() const;
void diameterRange(const ccRange& diameterRange);
```

- `ccRange diameterRange() const;`

  Returns the diameter range, in client units.

- `void diameterRange(const ccRange& diameterRange);`

  Sets the diameter range, in client units.

  At train time, all balls whose diameter lies within this range are identified. Some or all of these blobs are then trained as the model, depending on *diameterUncertainty*.

  Parameters

  diameterRange The diameter range.

  Throws

  BadParams

  The range is empty or invalid.

  The default is 0 to HUGE_VAL. It is not recommended to use this setting, as train time may be overly long and robustness is better when a more realistic range is specified.

  Notes

  Extraction diameter is somewhat approximate, so it may be necessary to pad this range slightly to extract all desired candidates.

  This parameter is ignored when using Geometric Training overloads of train.

diameterUncertainty

```
double diameterUncertainty() const;
void diameterUncertainty(double diameterUnc);
```

- `double diameterUncertainty() const;`

  Returns the diameter uncertainty, in client units.

- `void diameterUncertainty(double diameterUnc);`

  Sets the diameter uncertainty, in client units.

  This uncertainty is used to facilitate Image-based training, by picking the "most popular" blob size and training only those. The exact effect is to select the largest subset of found blobs that lie within this uncertainty of their mean diameter.

  If there is a tie on that metric, the group with the smallest spread (difference from smallest to largest) is selected from among the tied groups.

  Parameters

  diameterUnc The diameter uncertainty.

  Throws

  BadParams

  diameterUncertainty < 0

  The default is HUGE_VAL. This value is appopriate when not doing "autotrain" of blob size. This means to use all of the blobs in the *diameterRange* for training.

  Notes

  This parameter is ignored when using Geometric Training overloads of train.

polarity

```
ccBallPatternAlignDefs::Polarity polarity() const;
void polarity(ccBallPatternAlignDefs::Polarity pol);
```

- `ccBallPatternAlignDefs::Polarity polarity() const;`

  Returns the target blob polarity.

- `void polarity(ccBallPatternAlignDefs::Polarity pol);`

  Sets the target blob polarity.

  Parameters

  pol The target blob polarity.

  Throws

  BadParams

  Polarity setting is invalid.

Notes

This parameter is ignored when using Geometric Training overloads of train.

When *eUnknown* polarity is set, blobs of each polarity are extracted, and then the polarity that trains the most blobs, subject to *diameterRange* and *diameterUncertainty*, is chosen. Only one polarity is actually trained.

eUnknown polarity will generally double the train time.

refineMode

```
ccBallPatternAlignDefs::RefineMode refineMode() const;

void refineMode(
  ccBallPatternAlignDefs::RefineMode aRefineMode);
```

- `ccBallPatternAlignDefs::RefineMode refineMode() const;`

Returns the refinement mode.

- `void refineMode(`
  `ccBallPatternAlignDefs::RefineMode aRefineMode);`

Sets the refinement mode.

Refine mode guides the tool how to refine the pose and score at runtime in the following manner:

ccBallPatternAlignDefs::eRefineByImageFeatures:

Use features collected from the training image to refine (even if the tool is trained by geometric training).

ccBallPatternAlignDefs::eRefineByTrainedBlobs:

Use trained blobs (that is, what is returned by **ccBallPatternAlignModel::trainedBlobs()**) to refine.

Parameters

aRefineMode The refine mode.

Throws

BadParams

Refine mode setting is invalid.

The default is *ccBallPatternAlignDefs::kRefineDefault*

zoneEnable

```
c_UInt32 zoneEnable() const;
void zoneEnable(c_UInt32 enable);
```

- `c_UInt32 zoneEnable() const;`

  Returns the zone enable for each degree of freedom.

- `void zoneEnable(c_UInt32 enable);`

  Sets the zone enable for each degree of freedom.

  Non-zero bits in *enable* indicate degrees of freedom to be searched.

  If *zoneEnable* is set for a given DOF, **ccBallPatternAlignModel::train()** will train between *zoneLow* and *zoneHigh*. Otherwise, **ccBallPatternAlignModel::train()** will use the nominal value. “nominal” and “zone” are two separate train modes.

  Parameters

  enable The zone enable for each degree of freedom.

  Throws

  BadParams
  The given *enable* value was not:
  1) zero or
  2) created by a bitwise-OR of the values from *ccBallPatternAlignDefs::DOF*

  The default is 0 (no DOFs are enabled).

  Notes

  If *zoneEnable* is set for a given DOF, then the **nominal()** value is ignored.

nominal

```
double nominal(ccBallPatternAlignDefs::DOF dof) const;
void  nominal(ccBallPatternAlignDefs::DOF dof, double val);
```

- `double nominal(ccBallPatternAlignDefs::DOF dof) const;`

  Returns the nominal value for the given DOF.

  Parameters

  dof The given DOF.

- `void nominal(ccBallPatternAlignDefs::DOF dof, double val);`

Sets the nominal value for the given DOF.

The nominal value is only used if the given DOF is not enabled by **zoneEnable()**. The nominal value need not be related in any way to the values of **zoneLow()** and **zoneHigh()**.

Parameters

dof The given DOF.

val The nominal value to be set.

Throws

BadParams

The given nominal value for the given DOF is illegal. See the following table regarding what is legal.

The defaults:

Angle: 0.0.

Uniform, X, and Y scales: 1.0.

Notes

If a zone is enabled, then the nominal value is ignored.

zoneLow `double zoneLow (ccBallPatternAlignDefs::DOF dof) const;`

Returns the low zone parameter for the given DOF.

Parameters

dof The given DOF.

Throws

BadParams

The given DOF value is illegal. See the following table regarding what is legal.

For the defaults, see the table hereinafter.

zoneHigh `double zoneHigh (ccBallPatternAlignDefs::DOF dof) const;`

Returns the high zone parameter for the given DOF.

Parameters

dof The given DOF.

Throws

BadParams

The given DOF value is illegal. See the following table regarding what is legal.

For the defaults, see the table hereinafter.

zone

```
void  zone(ccBallPatternAlignDefs::DOF dof, double low,
 double high);
```

Sets the low and high zone parameters for the given DOF.

Parameters

dof The given DOF.

low The low zone.

high The high zone.

Throws

BadParams

Illegal low zone or high zone values are given for the given DOF. See the following table regarding what is legal.

For the defaults, see the table hereinafter.

DOF Summary Table

This table summarizes the degrees of freedom values for this tool.

DOF	Unit	Default low	Default high	Range	Restrictions
Angle	degrees	-45	+45	any	none
Uniform scale	multiplier	0.8	1.2	> 0	zoneLow is less than or equal to zoneHigh
X scale	multiplier	0.8	1.2	> 0	zoneLow is less than or equal to zoneHigh
Y scale	multiplier	0.8	1.2	> 0	zoneLow is less than or equal to zoneHigh

Notes

- Uniform scale is equal scale in both the x and y dimensions.
- X scale and Y scale only scale in the given dimension.

- Uniform scale is specified as a multiplier of the trained model size. For example, a uniform scale of 1.25 implies that the runtime (client coordinate) size of the model will be 25% larger in both dimensions than the trained model.
- The angle zone is always searched from the zoneLow to the zoneHigh. So, setting zoneLow to 0 and zoneHigh to 180 searches one half of the 360 degree angle range – setting zoneHigh to 0 and zoneLow to 180 searches the other half.

maxCoarseAlignGrainLimit

```
double maxCoarseAlignGrainLimit() const;
```

Returns the maximum coarse granularity limit for alignment.

maxFineAlignGrainLimit

```
double maxFineAlignGrainLimit() const;
```

Returns the maximum fine granularity limit for alignment.

maxRefineGrainLimit

```
double maxRefineGrainLimit() const;
```

Returns the maximum granularity limit for refinement.

maxGrainLimits

```
void maxGrainLimits(double maxCoarseAlignGrainLimit,
  double maxFineAlignGrainLimit,
  double maxRefineGrainLimit);
```

Sets the maximum granularity limits for alignment and refinement. At training time, the tool automatically determines the granularity limits, which are bounded by these maximum values.

Parameters

maxCoarseAlignGrainLimit
The maximum coarse granularity limit for alignment to be set.

maxFineAlignGrainLimit
The maximum fine granularity limit for alignment to be set.

maxRefineGrainLimit
The maximum granularity limit for refinement to be set.

Throws

BadParams

If the following condition is not met:
1 <= *maxRefineGrainLimit*
<= *maxFineAlignGrainLimit*
<= *maxCoarseAlignGrainLimit*

The defaults are as follows:

maxCoarseAlignGrainLimit: 40

maxFineAlignGrainLimit: 8

maxRefineGrainLimit: 2

ccBallPatternAlignTrainParams

ccBezierCurve

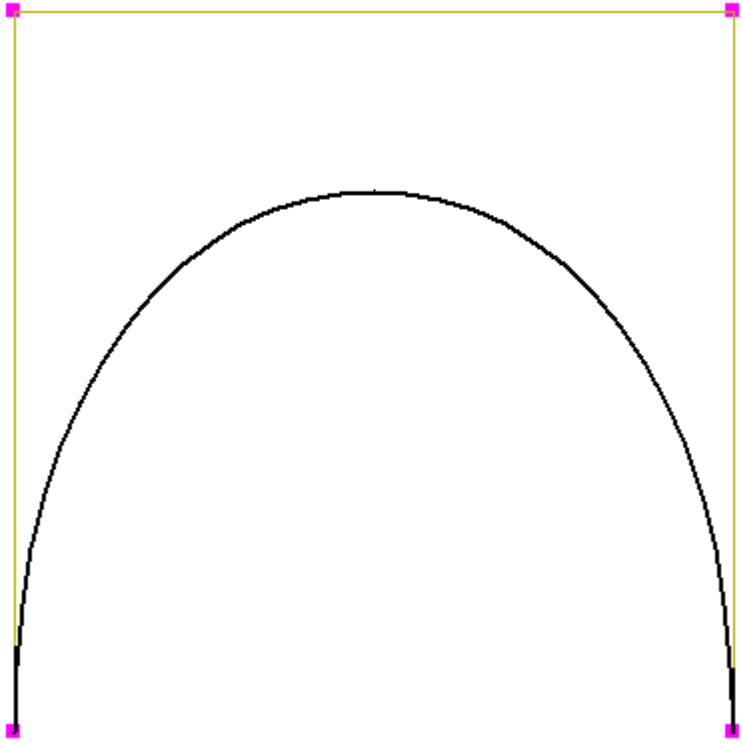
```
#include <ch_cvl/spline.h>

class ccBezierCurve : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccBezierCurve** class is an implementation of 2D cubic Bezier curves with optional control point weights.



The above figure shows an example of a Bezier curve with four control points.

Bezier Curves

A 2D cubic Bezier curve is a finite, continuous, and infinitely differentiable curve specified by four 2D Bezier control points. Bezier curves interpolate the first and last control points. It generally does not interpolate the inner two control points, although these affect the shape of the curve.

You can assign optional positive scalar weights to the control points to produce a *rational* Bezier curve. Coordinate functions along a rational Bezier curve are of the form:

$$x(t) = (f(t))/(h(t))$$

$$y(t) = (g(t))/(h(t))$$

where *f*, *g*, and *h* are cubic polynomials determined by the control points and weights. If all weights are equal, the function h(*t*) is a constant, and therefore x(*t*) and y(*t*) are themselves cubic polynomials. Weights are relative quantities, and multiplying all control point weights by a common factor thus would have no effect on the curve.

Parameter Range

Mathematically, the range of the parameter *t* is arbitrary, but all **ccBezierCurve** class methods that take or return parameter values assume a parameter range of 0 through 1, where 0 corresponds to the start point of the Bezier curve, and 1 corresponds to the endpoint. Note that the parameter *t* does not correspond to arc length. Sampling at uniformly spaced values of *t* does not in general produce points that are equally spaced with respect to arc length along the curve.

Tangent Vectors

Some methods of the Bezier curve and spline classes refer to tangent vectors. Unless otherwise noted, the implied meaning is the *parametric* tangent, given by [x'(*t*), y'(*t*)], where ' denotes differentiation with respect to *t*. The parametric tangent is subtly different from the geometric tangent to the curve at *t*. For example, if the first two control points of a Bezier curve are coincident, [x'(0), y'(0)], and so the parametric tangent vanishes. Yet the curve still has a geometric tangent at *t = 0* if the second or third derivatives of x(*t*) and y(*t*) do not all vanish. Also, the magnitude of the parametric tangent is not generally constant along a Bezier or spline curve. What is always true is that the parametric tangent at a given point on the curve is either zero or lies along the geometric tangent line to the curve at that point.

Note For in-depth information on the theory and use of 2D cubic Bezier curves and splines, see any textbook on the subject, such as *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, 1990, ISBN 0-12-249051-7.

Note All methods defined for this class leave a **ccBezierCurve** object unchanged when they throw an error.

Constructors/Destructors

```
ccBezierCurve();

explicit
ccBezierCurve(const cmStd vector<cc2Vect> &points);

explicit
ccBezierCurve(const cmStd vector<cc2Vect> &points,
  const cmStd vector<double> &weights);

explicit
ccBezierCurve(const cmStd vector<cc3Vect>
  &weightedControlPoints);
```

- `ccBezierCurve();`

  Default constructor. Constructs a default Bezier curve with all four control points coincident, equal to the origin, and with unit weights (that is, all weights equal to 1.0).

- ```
  explicit
  ccBezierCurve(const cmStd vector<cc2Vect> &points);
  ```

  Constructs a **ccBezierCurve** with the given control points and unit weights. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

  points The vector of control points.

  Notes

  The control points correspond to the Bezier curve's start point, middle two handle points, and end point, in that order.

  Throws

  ccShapesError::BadParams

  points does not have a size of four.

- ```
  explicit
  ccBezierCurve(const cmStd vector<cc2Vect> &points,
    const cmStd vector<double> &weights);
  ```

  Constructs a Bezier curve with the given control points and weights. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

  points The four 2D Bezier control points that specify the curve.

weights Positive scalar weights assigned to the control points to produce a *rational* Bezier curve (the default weight is 1.0 for all four points).

Notes

The elements of the vectors passed in correspond to the Bezier curve's start point, middle two handle points, and end point, in that order.

Throws

ccShapesError::BadParams
The *points* and *weights* vectors do not both have a size of 4.

ccShapesError::NonpositiveWeight
Any element of *weights* is non-positive.

- ```
  explicit
  ccBezierCurve(
    const cmStd vector<cc3Vect> &weightedControlPoints);
  ```

Constructs a Bezier curve with the given weighted control points. A control point (*x*, *y*) with weight *w* is represented as the **cc3Vect (w*x, w*y, w)**. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

weightedControlPoints
Vector of weighted control points.

Notes

The elements of the vector passed in correspond to the Bezier curve's start point, middle two handle points, and end point, in that order.

Throws

ccShapesError::BadParams
The *weightedControlPoints* vector does not have a size of 4.

ccShapesError::NonpositiveWeight
Any weight is non-positive.

Operators

operator== `bool operator==(const ccBezierCurve &rhs) const;`

Returns true if and only if this ccBezierCurve is equal to *rhs*. Two **ccBezierCurve**s are equal if their control points and weights are all identical (no tolerance is used).
  ```

Parameters

rhs — The other **ccBezierCurve**.

Notes

ccBezierCurves that describe identical curves are not necessarily equal. For example, the curve itself is invariant with respect to uniform scaling of weights, but **operator==()** is not.

operator!=

```
bool operator!=(const ccBezierCurve &rhs) const;
```

Returns the opposite truth value to **operator==()**.

Parameters

rhs — The other **ccBezierCurve**.

Public Member Functions

isWeighted

```
bool isWeighted() const;
```

Returns true if and only if this Bezier curve is weighted; that is, not all of the weights are equal.

controlPoint

```
cc2Vect controlPoint(c_Int32 idx) const;
void controlPoint(c_Int32 idx, const cc2Vect &ctrlPt);
```

- `cc2Vect controlPoint(c_Int32 idx) const;`

  Gets the control point with the given index.

  Parameters

  idx — The index.

- `void controlPoint(c_Int32 idx, const cc2Vect &ctrlPt);`

  Sets the control point with the given index.

  Parameters

  idx — The index.

  ctrlPt — The control point.

  Throws

  ccShapesError::BadIndex

  idx is either less than 0 or greater than or equal to 4.

controlPoints

```
cmStd vector<cc2Vect> controlPoints() const;
void controlPoints(const cmStd vector<cc2Vect> &points);
```

- `cmStd vector<cc2Vect> controlPoints() const;`

  Gets the vector of Bezier control points.

- `void controlPoints(const cmStd vector<cc2Vect> &points);`

  Sets the vector of Bezier control points.

  Parameters

  points The vector of four Bezier control points.

  Throws

  ccShapesError::BadParams
  The control points vector does not have a size of 4.

weight

```
double weight(c_Int32 idx) const;
void weight(c_Int32 idx, double newWeight);
```

- `double weight(c_Int32 idx) const;`

  Gets the weight of the control point with the given index.

  Parameters

  idx The index.

- `void weight(c_Int32 idx, double newWeight);`

  Sets the weight of the control point with given index.

  Parameters

  idx The index.

  newWeight The weight.

  Throws

  ccShapesError::BadIndex
  idx is either less than 0 or greater than or equal to 4.

  ccShapesError::NonpositiveWeight
  newWeight is non-positive.

weights

```
cmStd vector<double> weights() const;
void weights(const cmStd vector<double> &wghts);
```

- `cmStd vector<double> weights() const;`

  Gets the vector of control point weights.

- `void weights(const cmStd vector<double> &wghts);`

  Sets the vector of control point weights.

  Parameters

  wghts The vector of four control point weights.

  Throws

  ccShapesError::BadParams
  wghts vector does not have a size of 4.

  ccShapesError::NonpositiveWeight
  Any of the new weights are non-positive.

map

```
ccBezierCurve map(const cc2Xform& X) const;
```

Returns this Bezier curve mapped by *X*.

Parameters

X The 2D transform used to modify this Bezier curve.

Notes

This function maps each of the four control points by *X* while leaving the weights unchanged. The locus of points of the resulting mapped Bezier curve is exactly the locus of points of the original Bezier curve mapped by X.

point

```
cc2Vect point(double t) const;
```

Returns the point at parameter value *t* along the Bezier curve. Values of *t* outside of the range 0 through 1 are legal, but correspond to points on the extrapolation of the curve, not on the curve itself. The value returned is an exact value, not an approximation.

Parameters

t The parameter value.

tangent

```
cc2Vect tangent(double t) const;
```

Returns the tangent vector at parameter value *t* along the Bezier curve. Values of *t* outside of the range 0 through 1 are legal, but correspond to points on the extrapolation of the curve, not on the curve itself. The value returned is an exact value, not an approximation.

Parameters

t	The parameter value.

pointAndTangent

```
void pointAndTangent(double t, cc2Vect &point,
  cc2Vect &tangent) const;
```

Returns the point and tangent vector at parameter value *t* along the Bezier curve. Values of *t* outside of the range 0 through 1 are legal, but correspond to points on the extrapolation of the curve, not on the curve itself. The value returned is an exact value, not an approximation.

Parameters

t	The parameter value.

Notes

Calling this routine is more efficient than calling **point()** and **tangent()** separately.

nearestPoint

```
cc2Vect nearestPoint(const cc2Vect &p, double &t) const;

virtual cc2Vect nearestPoint(const cc2Vect& p) const;
```

- `cc2Vect nearestPoint(const cc2Vect &p, double &t) const;`

  Returns the nearest point on this Bezier curve to the given point as well as the nearest point's parameter value. If the nearest point is not unique, one of the nearest points is returned.

  Parameters

p	The point.
t	The nearest point's parameter value.

- `virtual cc2Vect nearestPoint(const c2Vect& p) const;`

  Returns the nearest point on this Bezier curve to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

p The point.

See **ccShape::nearestPoint()** for more information.

intersections

```
cmStd vector<double> intersections(const ccLineSeg &seg)
 const;
```

Returns the parameter values at which this Bezier curve intersects the given line segment.

Parameters

seg The line segment.

Notes

Returns an empty vector if there are no intersections.

There are infinite intersections if a portion of the spline lies exactly on the line segment. In this case, only a single intersection from this infinite set is returned.

subCurve

```
ccBezierCurve subCurve(double s, double t) const;
```

Returns the subcurve of this Bezier curve lying between the parameter values *s* and *t*.

Parameters

s The first parameter value.

t The second parameter value.

Notes

The cases $s == t$, $s > t$, and $s < t$ are all legal.

Passing $s = 1.0$ and $t = 0.0$ returns the same curve as **reverse()**.

The values of *s* and *t* are internally clipped if they fall outside the range 0 through 1.

cubicCoeffs

```
void cubicCoeffs(cmStd vector<double> &fCoeffs,
  cmStd vector<double> &gCoeffs,
  cmStd vector<double> &hCoeffs) const;
```

This Bezier curve is parameterized by

$$x(t) = (f(t))/(h(t))$$

$$y(t) = (g(t))/(h(t))$$

where *f*, *g*, and *h* are cubic polynomials. This method computes and returns these cubics. Each of the returned vectors has four elements, corresponding to the cubic coefficients, leading coefficient first.

Parameters

fCoeffs	Cubic coefficient of the polynomial f(*t*).
gCoeffs	Cubic coefficient of the polynomial g(*t*).
hCoeffs	Cubic coefficient of the polynomial h(*t*).

Notes

If **isWeighted()** is false, the polynomial h(*t*) is a constant.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this Bezier curve.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For Bezier curves, this function always returns true.

See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For Bezier curves, this function always returns false.

See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

Returns true if this shape has finite extent. For Bezier curves, this function always returns true.

See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For Bezier curves, this function always returns false.

See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

Returns true if the set of points that lie on the boundary of this shape is infinite, and there is a well-defined tangent at all but a finite number of these points. This function returns true for most Bezier curves. It returns false if all four control points are coincident, in which case the curve collapses to a single point and there is no tangent.

See **ccShape::hasTangent()** for more information.

isReversible

```
virtual bool isReversible() const;
```

Returns true if this shape can be reversed. For Bezier curves, this function always returns true.

See **ccShape::reverse()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

Returns true if this shape is already decomposed. For Bezier curves, this function always returns true.

See **ccShape::isDecomposed()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this Bezier curve.

Notes

The returned shape is a tight rectangle based on the curve itself, not the control points.

See **ccShape::boundingBox()** for more information.

perimeter

```
virtual double perimeter() const;
```

Returns the perimeter of this Bezier curve.

See **ccShape::perimeter()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this Bezier curve to the given point, as determined by **nearestPoint()**.

Parameters

info Shape information for this shape.

point The point.

See **ccShape::nearestPerimPos()** for more information.

startPoint

```
virtual cc2Vect startPoint() const;
```

Returns the starting point of this Bezier curve.

See **ccShape::startPoint()** for more information.

endPoint

```
virtual cc2Vect endPoint() const;
```

Returns the ending point of this Bezier curve.

See **ccShape::endPoint()** for more information.

startAngle

```
virtual ccRadian startAngle() const;
```

Returns the starting angle of this Bezier curve.

Throws

ccShapesError::NoTangent
hasTangent() is false for this Bezier curve.

See **ccShape::startAngle()** for more information.

endAngle

```
ccRadian endAngle() const;
```

Returns the ending angle for this Bezier curve.

Throws

ccShapesError::NoTangent
hasTangent() is false for this Bezier curve.

See **ccShape::endAngle()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this Bezier curve.

See **ccShape::reverse()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this Bezier curve mapped by *X*.

Parameters

X The transformation object.

See **ccShape::mapShape()** for more information.

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Returns the net signed angle through which the tangent vector rotates along this open contour shape from the start point to the end point.

Throws

ccShapesError::NoTangent
hasTangent() is false.

See **ccShape::tangentRotation()** for more information.

windingAngle

```
virtual ccRadian windingAngle(const cc2Vect &p) const;
```

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve.

Parameters

p The start point of the vector *p->t* whose angle is measured as the end point *t* traces the curve.

See **ccShape::windingAngle()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
 ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this **ccBezierCurve**. Returned sample points are generally not equally spaced along the curve.

Parameters

params	Parameters object specifying details of how the sampling should be done.
result	Result object to which position and tangent chains are stored.

Notes

If **params.computeTangents()** is true, this function ignores Bezier curves for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a clone of this Bezier curve. As a Bezier curve is already a decomposed shape, this method is equivalent to **clone()** for Bezier curves.

See **ccShape::decompose()** for more information.

subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this Bezier curve over the given perimeter range.

Parameters

info	Shape information for this Bezier curve.
range	The perimeter range.

See **ccShape::subShape()** for more information.

ccBlob

```
#include <ch_cvl/blobdesc.h>

class ccBlob: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that contains a single feature produced by performing blob analysis on an image.

ccBlob

```
ccBlob();

ccBlob(const ccBlob& rhs);
```

- `ccBlob();`

  You should not create **ccBlob**s.

Enumerations

Direction

`enum Direction`

This enumeration defines the direction codes that can appear in the chain code that describes the perimeter of a feature.

Value	Meaning
ePlusX	Increase in x-direction
eMinusX	Decrease in x-direction
ePlusY	Increase in y-direction
eMinusY	Decrease in y-direction
eEnd	End of chain

Measure `enum Measure`

This enumeration defines all the measures that can be computed for a feature.

Value	Meaning
eLabel	The feature label. For grey-scale connectivity this is 1 for blobs, 0 for holes. For labeled connectivity this is the feature's label.
eArea	The area of the feature in client coordinate system units. For grey-scale connectivity, this is the sum of the weights of the pixels. For other types of connectivity, it is the number of pixels.
eNumSteps	The number of steps in the chain code that describes this feature's perimeter
ePerimeter	The perimeter of the feature in client coordinate system units. This value is based on *eNumSteps* but is corrected to approximate the true perimeter of the feature. The correction formula is described as part of the **perimeter()** function on page 583.
eNumChildren	The number of features contained within this feature
eCenterMassX	The x-coordinate of the center of mass of the feature
eCenterMassY	The y-coordinate of the center of mass of the feature
eInertiaX	The second moment of inertia of this feature about an axis drawn through the feature's center of mass and parallel to the image coordinate system x-axis
eInertiaY	The second moment of inertia of this feature about an axis drawn through the feature's center of mass and parallel to the image coordinate system y-axis

Value	Meaning
eInertiaMin	The second moment of inertia of this feature about its major axis
eInertiaMax	The second moment of inertia of this feature about its minor axis
eElongation	The ratio of *eInertiaMax* to *eInertiaMin*
eAngle	The angle of the major axis of the feature with respect to the client coordinate system x-axis. If *eElongation* is 1.0 *eAngle* is reported as 0, although it is actually undefined.
eAcircularity	The acircularity of the feature as measured by dividing the square of the feature's perimeter by 4 times the feature's area times pi. The value of *eAcircularity* is 1 for a perfectly circular feature; the less circular the feature, the greater the value of *eAcircularity.*
eAcircularityRms	The acircularity of the feature as measured by the normalized rms deviation of the boundary point radius values from r0, where r0 is the square root of the feature's area divided by pi. The value of *eAcircularityRms* is 1 for a perfectly circular feature; the less circular the feature, the greater the value of *eAcircularityRms*.
eIsInterior	The blob is interior to the image.
eImageBoundCenterX	The x-coordinate of the center of the rectangle defining the image coordinate extents of the feature.
eImageBoundCenterY	The y-coordinate of the center of the rectangle defining the image coordinate extents of the feature

Value	Meaning
eImageBoundMinX	The x-coordinate of the left edge of the rectangle defining the image coordinate extents of this feature
eImageBoundMaxX	The x-coordinate of the right edge of the rectangle defining the image coordinate extents of this feature
eImageBoundMinY	The y-coordinate of the top edge of the rectangle defining the image coordinate extents of this feature
eImageBoundMaxY	The y-coordinate of the bottom edge of the rectangle defining the image coordinate extents of this feature
eImageBoundWidth	The width of the rectangle defining the image coordinate extents of this feature
eImageBoundHeight	The height of the rectangle defining the image coordinate extents of this feature
eImageBoundAspect	The ratio of *eImageBoundHeight* to *eImageBoundWidth*
eMedianX	The point on the client coordinate system x-axis through which a vertical line which divides the feature's area in half passes
eMedianY	The point on the client coordinate system y-axis through which a horizontal line which divides the feature's area in half passes
eBoundCenterX	The x-coordinate of the center of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription::extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription::extremaExcludeArea()**.

Value	Meaning
eBoundCenterY	The y-coordinate of the center of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription:: extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription:: extremaExcludeArea()**.
eBoundMinX	The x-coordinate of the left edge of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription:: extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription:: extremaExcludeArea()**.
eBoundMaxX	The x-coordinate of the right edge of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription:: extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription:: extremaExcludeArea()**.
eBoundMinY	The y-coordinate of the top edge of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription:: extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription:: extremaExcludeArea()**.

Value	Meaning
eBoundMaxY	The y-coordinate of the bottom edge of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription::extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription::extremaExcludeArea()**.
eBoundWidth	The width of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription::extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription::extremaExcludeArea()**.
eBoundHeight	The height of a bounding box aligned to the client coordinate system angle specified by the most recent call to **ccBlobSceneDescription::extremaAngle()** and respecting the amount of area to exclude specified by the most recent call to **ccBlobSceneDescription::extremaExcludeArea()**.
eBoundAspect	The ratio of *eBoundHeight* to *eBoundWidth*
eBoundPrincipalMinX	The distance, in client coordinates, along the first principal axis of the feature from the feature's center of mass in the negative X-direction to the edge of a bounding box aligned to the feature's principal axes. The distance is expressed in the modified client coordinate system returned by **ccBlob::boundingBoxPrincipal()**; the value of this measure is negative.

Value	Meaning
eBoundPrincipalMaxX	The distance, in client coordinates, along the first principal axis of the feature from the feature's center of mass in the positive X-direction to the edge of a bounding box aligned to the feature's principal axes. The distance is expressed in the modified client coordinate system returned by **ccBlob::boundingBoxPrincipal()**.
eBoundPrincipalMinY	The distance, in client coordinates, along the second principal axis of the feature from the feature's center of mass in the negative Y-direction to the edge of a bounding box aligned to the feature's principal axes. The distance is expressed in the modified client coordinate system returned by **ccBlob::boundingBoxPrincipal()**; the value of this measure is negative.
eBoundPrincipalMaxY	The distance, in client coordinates, along the second principal axis of the feature from the feature's center of mass in the positive Y-direction to the edge of a bounding box aligned to the feature's principal axes. The distance is expressed in the modified client coordinate system returned by **ccBlob::boundingBoxPrincipal()**.
eBoundPrincipalWidth	The distance, in client coordinates, along the first principal axis of the feature between the sides of the bounding rectangle aligned to the feature's first principal axis.

Value	Meaning
eBoundPrincipalHeight	The distance, in client coordinates, along the second principal axis of the feature between the sides of the bounding rectangle aligned to the feature's first principal axis.
eBoundPrincipalAspect	The ratio of *eBoundPrincipalHeight* to *eBoundPrincipalWidth*

Public Member Functions

scene

```
const ccBlobSceneDescription* scene() const;
```

Returns a pointer to the **ccBlobSceneDescription** of which this feature is a component.

id

```
c_Int32 id() const;
```

Returns the ID value of this feature. You can use this ID to refer to the feature.

label

```
c_UInt8 label() const;
```

Returns the type of feature that this **ccBlob** is. For grey-scale connectivity, this function returns 1 if the feature is a blob and 0 if it is a hole. For labelled connectivity, this function returns the feature's label value.

imageBoundingBox

```
ccPelRect imageBoundingBox() const;
```

Returns the smallest image-coordinate-aligned **ccPelRect** that encloses all of the pixels of this feature. The origin of the rectangle is the location of this blob in the original scene.

Notes

This function requires much less time to execute than functions that compute a feature's area or size with respect to its principal axes.

imageBoundingBoxCenter

```
cc2Vect imageBoundingBoxCenter() const;
```

Returns the location of the center of the **ccPelRect** returned by **imageBoundingBox()** in image coordinates.

Notes

This function requires much less time to execute than functions that compute a feature's area or size with respect to its principal axes.

imageBoundingBoxAspect

```
double imageBoundingBoxAspect() const;
```

Returns the ratio of the height of the **ccPelRect** returned by **imageBoundingBox()** to its width.

Notes

This function requires much less time to execute than functions that compute a feature's area or size with respect to its principal axes.

region

```
const ccRLEBuffer& region() const;
```

Returns a reference to a **ccRLEBuffer** that contains the feature described by this **ccBlob**.

If this **ccBlob** refers to a blob produced by grey-scale connectivity, runs in the returned **ccRLEBuffer** with nonzero values are part of the blob, while runs with a value of 0 are parts of holes or the background. If this **ccBlob** refers to a hole produced by grey-scale connectivity, runs in the returned **ccRLEBuffer** with a value of 1 are part of the hole, while runs with a value of 0 are parts of the background (either parts of the blob surrounding this hole or blobs within this hole). If this **ccBlob** refers to a feature produced by labelled connectivity, runs in the returned **ccRLEBuffer** with a value of 1 are part of the feature, while runs with a value of 0 are part of features with other labels.

boundaryChainCode

```
ccIPair boundaryChainCode(
  cmStd vector<ceBlobDirection>& chain) const;
```

Returns the location, in image coordinates, of the start (and end) of a chain code that describes this feature's outer boundary. The chain code itself is returned in the supplied reference to a vector of **ceBlobDirection**.

The chain code is defined as the path along the outside edges of the outermost pixels of the feature, moving in a clockwise direction. Each element of the chain code can indicate one of four directions. The end of the chain code is marked with an *eEnd* code.

Parameters

chain — The chain code. Each element of *chain* is one of the following values:

ePlusX
eMinusX
ePlusY
eMinusY
eEnd

Throws

ccBlob::BadMeasure
No chain code is available. This feature was produced using whole-image connectivity analysis.

numSteps

```
c_Int32 numSteps() const;
```

Returns the number of elements in the chain code that describes this feature's outer boundary.

Throws

ccBlob::BadMeasure
No chain code is available: this feature was produced using whole-image connectivity analysis.

area

```
double area() const;
```

Returns the area of the feature described by this **ccBlob** measured in client coordinate units. The area is computed in image coordinates, then converted to the units of the client coordinate system.

The initial area computation is performed as follows. If the feature is a blob that was produced by grey-scale connectivity, the area is the weighted sum of the nonzero pixels of the blob, normalized by any scaling value that you specified. If the feature is a hole produced by grey-scale connectivity, the area is the number of pixels in the hole. If the feature was produced by labelled connectivity, the area is the number of pixels in the feature.

Notes

The area is converted to client coordinate system units by calling the following functions:

```
region().clientFromImageXform().mapArea()
```

perimeter `double perimeter() const;`

Returns the perimeter of the feature. The perimeter is computed using the following formula:

$$P = 0.94806(Sx \times Nx + Sy \times Ny - Sc \times C)$$

Where:

Sx is the value returned by **cc2Xform::xScale()** when called on the **cc2Xform** associated with this feature.

Nx is the number of *ePlusX* and *eMinusX* codes in this feature's chain code.

Sy is the value returned by **cc2Xform::yScale()** when called on the **cc2Xform** associated with this feature.

Ny is the number of *ePlusY* and *eMinusY* codes in this feature's chain code.

Sc is equal to:

$$Sx + Sy - \sqrt{Sx^2 + Sy^2}$$

C is the number of convex corners in this feature's chain code. A convex corner is any of the following pairs of chain codes:

ePlusX followed by *eMinusY*
eMinusY followed by *eMinusX*
eMinusX followed by *ePlusY*
ePlusY followed by *ePlusX*

This formula corrects for the tendency of the raw chain code count to overstate the true blob perimeter.

The perimeter is returned in the units of the client coordinate system associated with the image being analyzed.

Throws

ccBlob::BadMeasure
No chain code is available: this feature was produced using whole-image connectivity analysis.

centerOfMass `cc2Vect centerOfMass() const;`

Returns the position of the center of mass of this feature in the client coordinate system.

inertia `ccDPair inertia() const;`

Returns the feature's second moments of inertia about the horizontal and vertical axes within the client coordinate system through the feature's center of mass.

inertia().x() contains the moment about the vertical axis, **inertia().y()** contains the moment about the horizontal axis.

inertiaPrincipal `ccDPair inertiaPrincipal() const;`

Returns the feature's second moments of inertia about its first and second principal axes.

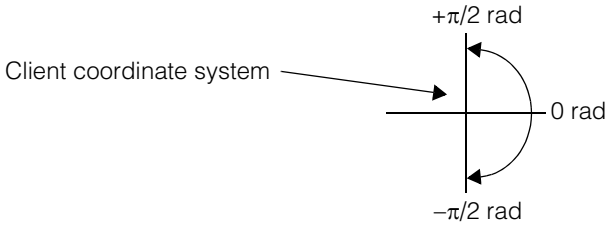
inertiaPrincipal().x() contains the moment about the first principal axis, **inertiaPrincipal().y()** contains the moment about the second principal axis.

elongation `double elongation() const;`

Returns the ratio of the feature's second moment of inertia about its second principal axis to the feature's second moment of inertia about its first principal axis.

angle `ccRadian angle() const;`

Returns the orientation of the feature's first principal axis with respect to the x-axis of the client coordinate system, in radians. The following figure shows how the angle is reported:



Notes

If the value returned by **elongation()** is exactly 1, then this function returns 0, although the actual value for the angle is undefined. If the value returned by **elongation()** is very close 1, the value for the angle may be meaningless.

acircularity

```
double acircularity() const;
```

Returns a measure of the acircularity of this feature. The return value is computed using the following formula:

$$p^2 / (4 \times \pi \times A)$$

Where:

p is the value returned by **perimeter()**

A is the value returned by **area()**

Throws

ccBlob::BadMeasure

No perimeter information is available: this feature was produced using whole-image connectivity analysis.

acircularityRms

```
double acircularityRms() const;
```

Returns a measure of the acircularity of this feature. The return value is the normalized RMS deviation of boundary point radius values from r0, where r0 computed using the following formula:

$$\sqrt{\frac{A}{\pi}}$$

Where:

A is the value returned by **area()**

Throws

ccBlob::BadMeasure

No perimeter information is available: this feature was produced using whole-image connectivity analysis.

median

```
cc2Vect median(const ccRadian& angle = ccRadian(0)) const;
```

Returns the median point for this feature. The median point is defined as the point through which a pair of perpendicular lines will divide the feature in half, vertically and horizontally. By default, the lines are aligned to the x- and y-axes of the client coordinate system. You can specify an optional angle at which to draw the horizontal line.

Parameters

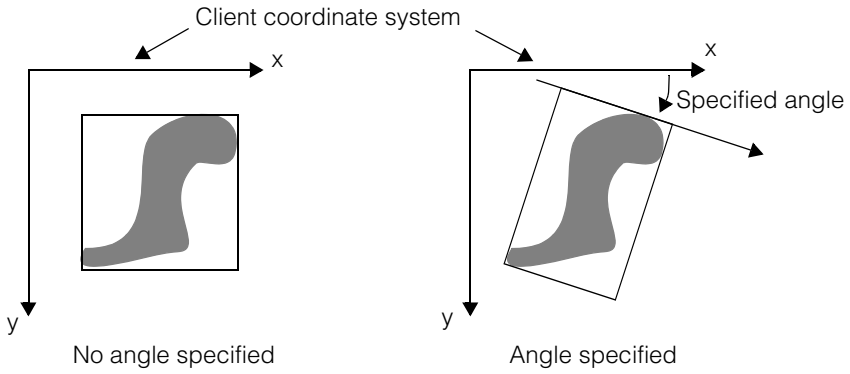
angle The angle (in the client coordinate system) at which to draw the horizontal median line. An angle of 0 is the same as the client coordinate system x-axis.

boundingBox

```
ccDRect boundingBox(bool excludeExtrema = true,
  const ccRadian& angle = ccRadian(0)) const;
```

Returns the smallest rectangle that completely encloses the feature described by this **ccBlob**. The bounding box is computed in a new coordinate system which is a rotated and translated version of the client coordinate system. The origin of the new coordinate system is the blob's center of mass, and the coordinate system is rotated by the specified angle, positive angles indicating that the new x-axis is rotated towards the +y-axis relative to the old x-axis.

The following diagram illustrates the effect of specifying a rotation angle:



You can specify the angle of the first principal axis of the feature to obtain the bounding box aligned to the blob's principal axis.

Parameters

excludeExtrema If *excludeExtrema* is *true*, the amount of the feature's extrema that you specified when you called **ccBlobSceneDescription::extremaExclude()**, **ccBlobSceneDescription::extremaExcludePels()**, or **ccBlobSceneDescription::extremaExcludePercent()** is excluded before the bounding box is computed.

If *excludeExtrema* is *false*, the bounding box is computed to enclose the entire feature.

angle The angle at which to align the bounding box, relative to the x-axis of client coordinate system.

Notes

Cognex recommends that you use the **boundingBox().origin()**, **boundingBox().width()**, and **boundingBox().height()** functions to obtain the bounding box dimensions.

The **boundingBox().origin()** function returns the location of the lower-left corner of the returned rectangle, in a coordinate system whose origin is at the center of the feature. Accordingly, the returned location will have negative values for both x and y.

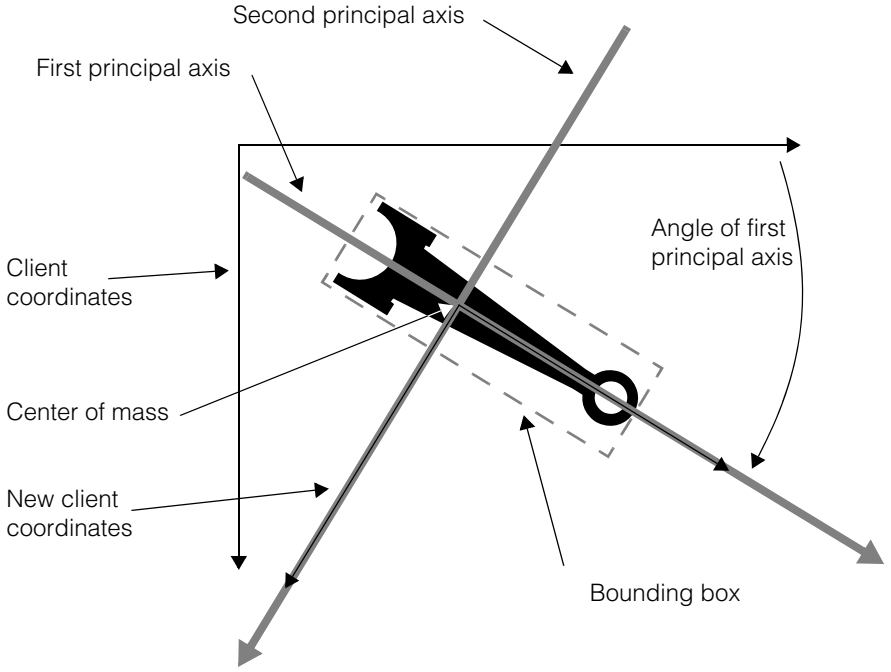
boundingBoxPrincipal

```
ccDRect boundingBoxPrincipal(bool excludeExtrema = true)
  const;
```

Returns the smallest rectangle that completely encloses the feature described by this **ccBlob** aligned to the feature's first principal axis. The returned **ccDRect** has its origin set to the center of mass of the feature.

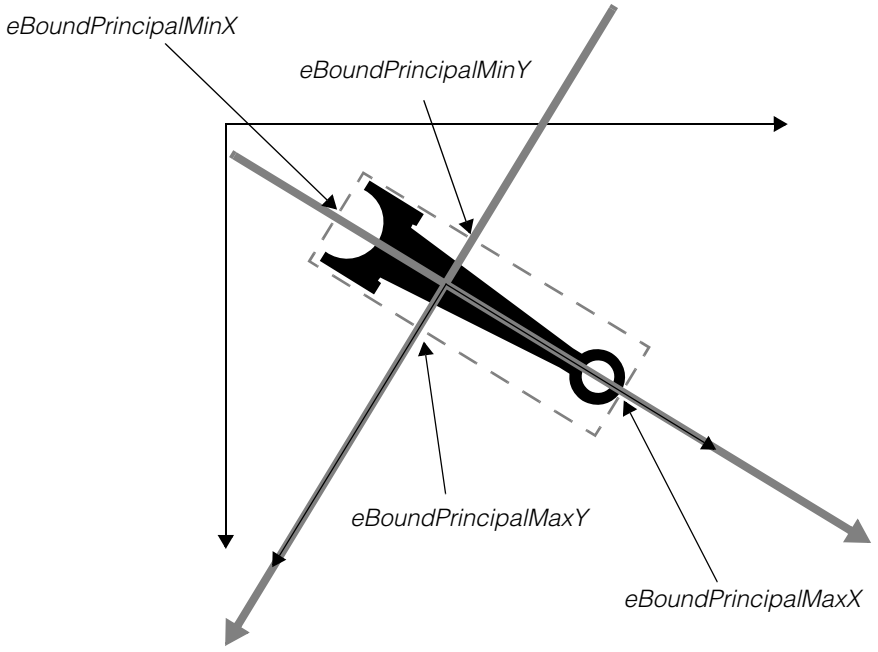
Unlike **imageBoundingBox()**, this function returns a measure of the blob that does not vary with the angle of the blob.

The location of the rectangle returned by this function is given in terms of a new client coordinate system. This new client coordinate system is a copy of the current client coordinates translated and rotated such that the origin of the new coordinate system is at the feature's center of mass and the positive x-axis is aligned to the first principal axis of the feature. The following figure shows how this new coordinate system is created:



The blob measures *eBoundPrincipalMinX*, *eBoundPrincipalMaxX*, *eBoundPrincipalMinY*, and *eBoundPrincipalMaxY* are all expressed using this same

client coordinate system, as shown in the following figure:



Parameters

excludeExtrema If *excludeExtrema* is *true*, the amount of the feature's extrema that you specified when you called **ccBlobSceneDescription::extremaExclude()**, **ccBlobSceneDescription::extremaExcludePels()**, or **ccBlobSceneDescription::extremaExcludePercent()** is excluded before the bounding box is computed.

If *excludeExtrema* is *false*, the bounding box is computed to enclose the entire feature.

Notes

This function is equivalent to calling **boundingBox()** and specifying **angle()** for the *angle* argument.

Cognex recommends that you use the **boundingBoxPrincipal().origin()**, **boundingBoxPrincipal().width()**, and **boundingBoxPrincipal().height()** functions to obtain the bounding box dimensions.

center

```
cc2Vect center(bool excludeExtrema = true,
  const ccRadian& angle = ccRadian(0)) const;
```

Returns the position of the center of the bounding box computed by **boundingBox()**. The position is returned in terms of the client coordinate system for the image which contained this feature.

Parameters

excludeExtrema If *excludeExtrema* is *true*, the amount of the feature's extrema that you specified when you called **ccBlobSceneDescription::extremaExclude()**, **ccBlobSceneDescription::extremaExcludePels()**, or **ccBlobSceneDescription::extremaExcludePercent()** is excluded before the bounding box is computed.

If *excludeExtrema* is *false*, the bounding box is computed to enclose the entire feature.

angle The angle at which to align the bounding box, relative to the x-axis of client coordinate system.

centerPrincipal

```
cc2Vect centerPrincipal(bool excludeExtrema = true) const;
```

Returns the position of the center of the bounding box computed by **boundingBoxPrincipal()**. The position is returned in terms of the client coordinate system for the image which contained this feature.

Parameters

excludeExtrema If *excludeExtrema* is *true*, the amount of the feature's extrema that you specified when you called **ccBlobSceneDescription::extremaExclude()**, **ccBlobSceneDescription::extremaExcludePels()**, or **ccBlobSceneDescription::extremaExcludePercent()** is excluded before the bounding box is computed.

If *excludeExtrema* is *false*, the bounding box is computed to enclose the entire feature.

aspect

```
double aspect(bool excludeExtrema = true,
  const ccRadian& angle = ccRadian(0)) const;
```

Returns the aspect ratio of the bounding box computed by **boundingBox()**. This is equal to the ratio of the value returned by **ccDRect::height()** to the value returned by **ccDRect::width()** when called on the **ccDRect** returned by **boundingBox()**.

Parameters

excludeExtrema If *excludeExtrema* is *true*, the amount of the feature's extrema that you specified when you called **ccBlobSceneDescription::extremaExclude()**, **ccBlobSceneDescription::extremaExcludePels()**, or **ccBlobSceneDescription::extremaExcludePercent()** is excluded before the bounding box is computed.

If *excludeExtrema* is *false*, the bounding box is computed to enclose the entire feature.

angle The angle at which to align the bounding box, relative to the x-axis of client coordinate system.

isInterior

```
bool isInterior() const;
```

This function tests whether the blob is interior to the image. The function may, or may not use the mask **ccBlobParams::mask()**.

If **ccBlobParams::useMaskForInterior()** is true and the mask is not degenerate, the mask is used. Otherwise the mask is not used.

When the mask is not used, the function returns false if some blob pixel is a boundary pixel of the input image. When the mask is used the function returns false if some blob pixel is 8-connected to some don't care pixel of the mask. In all other cases the function returns true.

aspectPrincipal

```
double aspectPrincipal(bool excludeExtrema = true) const;
```

Returns the aspect ratio of the bounding box computed by **boundingBoxPrincipal()**. This is equal to the ratio of the value returned by **ccDRect::height()** to the value returned by **ccDRect::width()** when called on the **ccDRect** returned by **boundingBoxPrincipal()**.

Parameters

excludeExtrema If *excludeExtrema* is *true*, the amount of the feature's extrema that you specified when you called **ccBlobSceneDescription::extremaExclude()**, **ccBlobSceneDescription::extremaExcludePels()**, or **ccBlobSceneDescription::extremaExcludePercent()** is excluded before the bounding box is computed.

If *excludeExtrema* is *false*, the bounding box is computed to enclose the entire feature.

Notes

The value returned by this function is typically less than or equal to 1 while the value returned by **elongation()** is always greater than or equal to 1.

measure

```
double measure(theMeasure) const;
```

Returns the specified measure for this feature. Measures are always computed and returned in terms of the client coordinate system.

Parameters

theMeasure The measure to obtain. *theMeasure* must be one of the following values:

ccBlob::eLabel
ccBlob::eArea
ccBlob::eNumSteps
ccBlob::ePerimeter
ccBlob::eNumChildren
ccBlob::eCenterMassX
ccBlob::eCenterMassY
ccBlob::eInertiaX
ccBlob::eInertiaY
ccBlob::eInertiaMin
ccBlob::eInertiaMax
ccBlob::eElongation
ccBlob::eAngle
ccBlob::eAcircularity
ccBlob::eAcircularityRms
ccBlob::eIsInterior
ccBlob::eImageBoundCenterX
ccBlob::eImageBoundCenterY
ccBlob::eImageBoundMinX
ccBlob::eImageBoundMaxX
ccBlob::eImageBoundMinY
ccBlob::eImageBoundMaxY
ccBlob::eImageBoundWidth
ccBlob::eImageBoundHeight
ccBlob::eImageBoundAspect
ccBlob::eMedianX
ccBlob::eMedianY
ccBlob::eBoundCenterX
ccBlob::eBoundCenterY
ccBlob::eBoundMinX
ccBlob::eBoundMaxX
ccBlob::eBoundMinY
ccBlob::eBoundMaxY

ccBlob::eBoundWidth
ccBlob::eBoundHeight
ccBlob::eBoundAspect
ccBlob::eBoundPrincipalMinX
ccBlob::eBoundPrincipalMaxX
ccBlob::eBoundPrincipalMinY
ccBlob::eBoundPrincipalMaxY
ccBlob::eBoundPrincipalWidth
ccBlob::eBoundPrincipalHeight
ccBlob::eBoundPrincipalAspect

Notes

This function is called internally by the Blob tool if you have defined sorting or filtering criteria in the **ccBlobSceneDescription** that contains this feature.

Throws

ccBlob::BadMeasure
The requested measure is not available: this feature was produced using whole-image connectivity analysis.

parent

```
const ccBlob* parent() const;
```

Returns a pointer to the feature that encloses this feature. If no feature encloses this one, this function returns *NULL*.

nextSibling

```
const ccBlob* nextSibling(bool filtered = true) const;
```

Returns a pointer to the next feature that is contained within this feature's parent. The order in which features are returned by this function is determined by the current sort order, which you can set by calling **scene().setSort()**. If this feature has no siblings, this function returns *NULL*.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

prevSibling

```
const ccBlob* prevSibling(bool filtered = true) const;
```

Returns a pointer to the previous feature that is contained within this feature's parent. The order in which features are returned by this function is determined by the current sort order, which you can set by calling **scene().setSort()**. If this feature has no siblings, this function returns *NULL*.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

numChildren

```
c_Int32 numChildren(bool filtered=true) const;
```

Returns the number of features that are enclosed within this feature.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

firstChild

```
const ccBlob* firstChild(bool filtered = true) const;
```

Returns a pointer to the first feature that is contained within this feature. The order in which features are returned by this function is determined by the current sort order, which you can set by calling **scene().setSort()**. If this feature does not contain any features, this function returns *NULL*.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

lastChild

```
const ccBlob* lastChild(bool filtered = true) const;
```

Returns a pointer to the last feature that is contained within this feature. The order in which features are returned by this function is determined by the current sort order, which you can set by calling **scene().setSort()**. If this feature does not contain any features, this function returns *NULL*.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

draw

```
void draw(ccGraphicList& graphList,
  c_UInt32 drawMode=ccBlobDefs::eDrawStandard,
  const ccCvlString& label=ccCvlString()) const;
```

Appends result graphics for this blob to the supplied **ccGraphicList**.

The graphics are drawn in client coordinates.

Parameters

graphList	The graphics list to append the graphics to.
drawMode	The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values: *ccBlobDefs::eDrawImageCenterOfMass* *ccBlobDefs::eDrawCenterOfMass* *ccBlobDefs::eDrawLabel* *ccBlobDefs::eDrawImageBoundingBox* *ccBlobDefs::eDrawBoundingBox* *ccBlobDefs::eDrawBoundingTrace* *ccBlobDefs::eDrawStandard*
label	If you include *ccBlobDefs::eDrawLabel* in *drawMode*, then the contents of *label* are used to label the center of mass.

ccBlobDefs

```
#include <blobdesc.h>

class ccBlobDefs;
```

A name space that holds enumerations and constants used with the Blob tool.

Enumerations

DrawMode

```
enum DrawMode
```

This enumeration defines the types of result graphics you can draw for blobs and blob scene descriptions.

Value	Meaning
eDrawImageCenterOfMass	Draws a cross at the center of mass.
eDrawCenterOfMass	Draws a cross at the center of mass aligned to the angle of the principal axis. (Used with *eDrawBoundingBox*.)
eDrawLabel	Draws a text label at the center of mass.
eDrawImageBoundingBox	Draws a box that encloses all of the feature's pixels. The box is image-aligned.
eDrawBoundingBox	Draws a box that encloses all of the feature's pixels. The box is aligned to the angle of the principal axis.
eDrawBoundingTrace	Draws the bounding trace (feature perimeter) that describes this feature.
eDrawStandard	Draws an image-aligned bounding box, image-aligned cross at the center of mass, and a text label at the center of mass.

Notes

All graphics are drawn in **ccColor::greenColor()** by default, except for *eDrawBoundingTrace*, which is drawn in **ccColor::cyanColor()**.

ccBlobDefs

ccBlobParams

```
#include <ch_cvl/blobtool.h>

class ccBlobParams: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains all of the parameters required to perform blob analysis on an image.

Constructors/Destructors

ccBlobParams `ccBlobParams();`

Constructs a **ccBlobParams** with the following default values:

Parameter	Default value
segmentationType	*eNone* Assumes a binary segmented image
scaleVal	1 Assumes a binary segmented image
keepRLE	False
mask	Degenerate (none)
keepMasked	False
morph	0-sized (none)
connectivityType	eGreyScale
connectivityCleanup	*eNone*
connectivityMinPels	0
useMaskForInterior	False

Each of these parameters may be examined and changed using **ccBlobParams** member functions.

Enumerations

MorphOp `enum MorphOp`

This enumeration defines the grey-scale morphological operations that are supported as part of the image segmentation process.

Value	Meaning
eMinH	Replace the target pixel with the minimum value of the 3 horizontal pixels centered on the target pixel.
eMinV	Replace the target pixel with the minimum value of the 3 vertical pixels centered on the target pixel.

Value	Meaning
eMinS	Replace the target pixel with the minimum value of the 9 square pixel area centered on the target pixel.
eMaxH	Replace the target pixel with the maximum value of the 3 horizontal pixels centered on the target pixel.
eMaxV	Replace the target pixel with the maximum value of the 3 vertical pixels centered on the target pixel.
eMaxS	Replace the target pixel with the maximum value of the 9 square pixel area centered on the target pixel.
eMinMaxH	Perform an *eMaxH* followed by an *eMinH*; also called closing.
eMinMaxV	Perform an *eMaxV* followed by an *eMinV*; also called closing.
eMinMaxS	Perform an *eMaxS* followed by an *eMinS* also called closing.
eMaxMinH	Perform an *eMinH* followed by an *eMaxH*; also called opening.
eMaxMinV	Perform an *eMinV* followed by an *eMaxV*; also called opening.
eMaxMinS	Perform an *eMinS* followed by an *eMaxS*; also called opening.

Segmentation `enum Segmentation`

This enumeration defines the segmentation types supported in **ccBlobParams**. You can only set the segmentation type by calling a member function that includes all of the parameters required for that segmentation type.

Value	Meaning
eNone	No segmentation is performed; the input image is already segmented.
eMap	Segment using a pixel map.
eHardThresh	Segment using a hard binary threshold; you can specify an absolute or relative threshold value.
eSoftThresh	Segment using a soft binary threshold; you can specify an absolute or relative threshold value.
eThreshImage	Segment using a threshold image.

Public Member Functions

setSegmentationNone

```
void setSegmentationNone(c_UInt8 scaleVal);
```

Sets this **ccBlobParams** to perform no segmentation on the input image. Use this function if the input image is already segmented.

You specify a scaling value to indicate the pixel value in the input image that represents a weight of 1.0. All pixels in the input image are interpreted on a linear scale relative to the scaling value. No pixel in the input image should have a value greater than *scaleVal*.

Parameters

scaleVal The scaling value.

Throws

ccBlobDefs::BadParams
scaleVal is equal to zero.

setSegmentationMap

```
void setSegmentationMap(const cmStd vector<c_UInt8>& map,
 c_UInt8 scaleVal);
```

Sets this **ccBlobParams** to segment the input image using the supplied pixel map. Each pixel in the input image is assigned a weight equal to the value within the pixel map at the index equal to the input image pixel value.

You specify a scaling value to indicate the pixel value in the resulting mapped image that represents a weight of 1.0. All pixels in the mapped image are interpreted on a linear scale relative to the scaling value; no pixel in the input image should have a value greater than *scaleVal*.

Parameters

map The pixel map. *map* must have at least as many elements as the largest pixel value in the input image.

scaleVal The scaling value.

Notes

This function creates an internal copy of the supplied pixel map.

Throws

ccBlobDefs::BadParams
scaleVal is equal to zero.
map size is equal to zero.

setSegmentationHardThresh

```
void setSegmentationHardThresh(c_UInt8 thresh,
  bool invert=false);

void setSegmentationHardThresh(c_Int32 lowTailPercent,
  c_Int32 highTailPercent, c_Int32 threshPercent,
  bool invert = false);
```

- ```
  void setSegmentationHardThresh(c_UInt8 thresh,
    bool invert=false);
  ```

  Sets this **ccBlobParams** to segment the input image using the supplied hard threshold. Each pixel in the input image with a value below the threshold is assigned a weight of 0; each pixel in the input image with a value greater than or equal to the threshold is assigned a weight of 1.

  You can specify an invert flag to specify that pixels below the threshold are assigned a weight of 1 and pixels at or above the threshold are assigned a weight of 0.

  Parameters

thresh	The threshold value.
invert	If *invert* is *true*, then all pixels in *image* with values greater than or equal to the supplied threshold are set to 0; pixels with values less than the threshold are set to 1. If *invert* is *false*, then all pixels in *image* with values greater than or equal to the supplied threshold are set to 1; pixels with values less than the threshold are set to 0.

- ```
  void setSegmentationHardThresh(c_Int32 lowTailPercent,
    c_Int32 highTailPercent, c_Int32 threshPercent,
    bool invert = false);
  ```

  Sets this **ccBlobParams** to segment the input image using the supplied relative threshold value. The specified percentage of low tail and high tail pixels are discarded, then the threshold pixel value is computed by determining the pixel value which lies at the specified percentage of the distance between the low tail pixel value and the high tail pixel value.

  Each pixel in the input image with a value below the computed threshold is assigned a weight of 0; each pixel in the input image with a value greater than or equal to the computed threshold is assigned a weight of 1.

  You can specify an invert flag to specify that pixels below the computed threshold are assigned a weight of 1 and pixels at or above the threshold are assigned a weight of 0.

Parameters

lowTailPercent The percentage of low tail pixels to discard before computing the threshold.

highTailPercent The percentage of pixels above which to discard high tail pixels before computing the threshold.

threshPercent The pixel value that lies *threshPercent* of the distance between the low tail pixel value and the high tail pixel value is used as the threshold value.

invert If *invert* is *true*, then all pixels in *image* with values greater than or equal to the computed threshold are set to 0; pixels with values less than the threshold are set to 1. If *invert* is *false*, then all pixels in *image* with values greater than or equal to the computed threshold are set to 1; pixels with values less than the threshold are set to 0.

Throws

ccBlobDefs::BadParams
Any percentage is not in the range 0 through 100, inclusive.

setSegmentationSoftThresh

```
void setSegmentationSoftThresh(c_UInt8 loThresh,
  c_UInt8 hiThresh, c_UInt8 softness, bool invert=false);

void setSegmentationSoftThresh(c_Int32 lowTailPercent,
  c_Int32 highTailPercent, c_Int32 lowThreshPercent,
  c_Int32 highThreshPercent, c_UInt8 softness,
  bool invert = false);
```

- ```
  void setSegmentationSoftThresh(c_UInt8 loThresh,
    c_UInt8 hiThresh, c_UInt8 softness, bool invert=false);
  ```

  Sets this **ccBlobParams** to segment the input image using the supplied soft threshold values.

  Each pixel in the input image with a value below the low threshold is assigned a weight of 0; each pixel in the input image with a value greater than or equal to the high threshold is assigned a weight of *softness*+1. The range of pixel values between the low and high threshold is divided into *softness* equally sized ranges. Pixels with values in each of these ranges are assigned values between 1 and *softness*.

  You can specify an invert flag to specify that pixels below the computed low threshold are assigned a weight of *softness*+1 and pixels at or above the computed high threshold are assigned a weight of 0.
  ```

Parameters

loThresh The low threshold. Pixels with values below *loThresh* are assigned a value of 0.

hiThresh The high threshold. Pixels with values greater than or equal to *hiThresh* are assigned a value of *softness*+1.

softness The number of intermediate pixel weights. The range of *loThresh* to *hiThresh* is divided into *softness* ranges. Pixels in each range receive a value equal to the number of the range in which they are.

softness should be in the range [1, *hiThresh* - *loThresh*]. Zero can be used, and is equivalent to hard thresholding with *hiThresh*. Higher *softness* values provide more accurate results, but processing may be slower than with a lower *softness* value.

invert If *invert* is *true*, then pixels with values greater than or equal to the high threshold are set to 0; pixels with values less than the low threshold are set to *softness*+1. If *invert* is *false*, then pixels with values greater than or equal to the high threshold are set to *softness*+1; pixels with values less than the low threshold are set to 0.

Throws

ccBlobDefs::BadParams

softness is equal to 255.
loThresh is greater than or equal to *hiThresh*.
softness is greater than (*hiThresh* minus *loThresh*).

- ```
  void setSegmentationSoftThresh(c_Int32 lowTailPercent,
    c_Int32 highTailPercent, c_Int32 lowThreshPercent,
    c_Int32 highThreshPercent, c_UInt8 softness,
    bool invert = false);
  ```

  Sets this **ccBlobParams** to segment the input image using the supplied soft threshold parameters. The specified percentage of low tail and high tail pixels are discarded, then two threshold pixel values are computed by determining the pixel values which lie at the specified percentages of the distance between the low tail pixel value and the high tail pixel value.

  Each pixel in the input image with a value below the computed low threshold is assigned a weight of 0; each pixel in the input image with a value greater than or equal to the computed high threshold is assigned a weight of *softness*+1. The range of pixel values
  ```

between the computed low and computed high threshold is divided into *softness* equally sized ranges. Pixels with values in each of these ranges are assigned values between 1 and *softness*.

You can specify an invert flag to specify that pixels below the computed low threshold are assigned a weight of *softness*+1 and pixels at or above the computed high threshold are assigned a weight of 0.

Parameters

lowTailPercent The percentage of low tail pixels to discard before computing the threshold.

highTailPercent The percentage of pixels above which to discard high tail pixels before computing the threshold.

lowThreshPercent

The pixel value that lies *lowThreshPercent* of the distance between the low tail pixel value and the high tail pixel value is used as the computed low threshold value.

highThreshPercent

The pixel value that lies *highThreshPercent* of the distance between the low tail pixel value and the high tail pixel value is used as the computed high threshold value.

softness The number of intermediate pixel weights. The range between the computed low and high thresholds is divided into *softness* ranges. Pixels in each range receive a value equal to the number of the range in which they are.

softness should be in the range [1, *hiThresh* - *loThresh*], where *hiThresh* and *loThresh* are the pixel values calculated from *lowThreshPercent* and *highThreshPercent*. If you believe you have images where these computed thresholds may lie close together, use a small *softness* value such as 1, 2, or 3.

invert If *invert* is *true*, then pixels with values greater than or equal to the computed high threshold are set to 0; pixels with values less than the computed low threshold are set to *softness*+1. If *invert* is *false*, then pixels with values greater than or equal to the computed high threshold are set to *softness*+1; pixels with values less than the computed low threshold are set to 0.

Notes

lowTailPercent must be less than *highTailPercent*, *lowThreshPercent* must be less than *highTailPercent*, and *softness* must be greater than or equal to 0.

Throws

ccBlobDefs::BadParams

softness is equal to 255.
Any percentage is not in the range of 0 through 100 inclusive.
lowTailPercent plus *highTailPercent* is greater than 100.
lowThreshPercent is greater than *highThreshPercent*.

setSegmentationThreshImage

```
void setSegmentationThreshImage(
  const cmStd vector<c_UInt8>& preMap,
  const ccPelBuffer_const<c_UInt8>& threshImg,
  const cmStd vector<c_UInt8>& postMap, c_UInt8 scaleVal);
```

Sets this **ccBlobParams** to segment the input image using the supplied threshold image and threshold image parameters.

The supplied pixel map is applied to the input image, then each pixel in the threshold image is subtracted from the corresponding pixel in the input image. A second pixel map is applied to the results of this subtraction.

You specify a scaling value to indicate the pixel value in the resulting image that represents a weight of 1.0. All pixels in the resulting image are interpreted on a linear scale relative to the scaling value; no pixel in the image should have a value greater than *scaleVal*.

Parameters

preMap — The pixel map to apply to the input image. Each pixel in the input image is used as an index into *preMap*. The input pixel value is replaced with the value from *preMap*.

preMap must be at least as large as the largest pixel value in the input image.

threshImg — The threshold image. *threshImg* must have the same dimensions as the input image. The value of each pixel in *threshImg* is subtracted from the corresponding pixel in the input image with the result clamped to 0. The image that results from this operation is mapped using *postMap*.

postMap — The pixel map to apply to the result of the image subtraction. Each pixel value that results from the subtraction of the *threshImg* from the input image is used as an index into *postMap*. The pixel value in the encoded image is the value found at that pixel value index in *postMap*.

postMap must be at least as large as the largest difference in pixel values between the input image and *threshImg*.

scaleVal The pixel value in the resulting image that represents a weight of 1.0. All pixels in the resulting image are interpreted on a linear scale relative to *scaleVal*; no pixel in *postMap* should have a value greater than *scaleVal*.

Notes

This function makes copies of *preMap*, *thresh*, and *postMap*. *threshImg* must be bound and located and is not copied. Any change to the pixels in *threshImg*'s root image will affect the segmentation.

Throws

ccBlobDefs::BadParams

scaleVal is equal to zero.
threshImg is unbound.

segmentationType

```
Segmentation segmentationType() const;
```

Returns the segmentation type of this **ccBlobParams**, one of the following values:

ccBlobParams::eNone
ccBlobParams::eMap
ccBlobParams::eHardThresh
ccBlobParams::eSoftThresh
ccBlobParams::eThreshImage

map

```
const cmStd vector<c_UInt8>& map() const;
```

If you called **setSegmentationMap()** to configure this **ccBlobParams**, this function returns the pixel map you specified.

If you called **setSegmentationThreshImage()** to configure this **ccBlobParams**, this function returns the pixel pre-map you specified.

Throws

ccBlobDefs::BadParams

Segmentation type is neither *ccBlobParams::eMap* nor *ccBlobParams::eThreshImage*.

postMap

```
const cmStd vector<c_UInt8>& postMap() const;
```

If you called **setSegmentationThreshImage()** to configure this **ccBlobParams**, this function returns the pixel post-map you specified.

Throws

ccBlobDefs::BadParams

Segmentation type is not *ccBlobParams::eThreshImage*.

thresh

```
c_UInt8 thresh() const;
```

If you called **setSegmentationHardThresh()** to configure this **ccBlobParams**, this function returns the threshold value.

If you called **setSegmentationSoftThresh()** to configure this **ccBlobParams**, this function returns the low threshold value.

Throws

ccBlobDefs::BadParams

Segmentation type is neither *ccBlobParams::eHardThresh* nor *ccBlobParams::eSoftThresh*.

hiThresh

```
c_UInt8 hiThresh() const;
```

If you called **setSegmentationSoftThresh()** to configure this **ccBlobParams**, this function returns the high threshold value.

Throws

ccBlobDefs::BadParams

Segmentation type is not *ccBlobParams::eSoftThresh*.

invert

```
bool invert() const;
```

If you called **setSegmentationSoftThresh()** or **setSegmentationHardThresh()** to configure this **ccBlobParams**, this function returns the value of the invert flag.

Throws

ccBlobDefs::BadParams

Segmentation type is neither *ccBlobParams::eHardThresh* nor *ccBlobParams::eSoftThresh*.

softness

```
c_UInt8 softness() const;
```

If you called **setSegmentationSoftThresh()** to configure this **ccBlobParams**, this function returns the softness value.

Throws

ccBlobDefs::BadParams

Segmentation type is not *ccBlobParams::eSoftThresh*.

isThreshPercent `bool isThreshPercent() const;`

If you called **setSegmentationSoftThresh()** or **setSegmentationHardThresh()** to configure this **ccBlobParams**, this function returns *true* if you specified relative (percentage) thresholds, *false* if you specified absolute (pixel value) thresholds.

Throws

ccBlobDefs::BadParams
Segmentation type is neither *ccBlobParams::eHardThresh* nor *ccBlobParams::eSoftThresh*.

lowTailPercent `c_Int32 lowTailPercent() const;`

If you called **setSegmentationSoftThresh()** or **setSegmentationHardThresh()** to configure this **ccBlobParams** and specified relative (percentage) thresholds, this function returns the low tail percentage value you specified.

Throws

ccBlobDefs::BadParams
Segmentation type is neither *ccBlobParams::eHardThresh* nor *ccBlobParams::eSoftThresh*.
Segmentation was not done with percentages.

hiTailPercent `c_Int32 hiTailPercent() const;`

If you called **setSegmentationSoftThresh()** or **setSegmentationHardThresh()** to configure this **ccBlobParams** and you specified relative (percentage) thresholds, this function returns the high tail percentage value you specified.

Throws

ccBlobDefs::BadParams
Segmentation type is neither *ccBlobParams::eHardThresh* nor *ccBlobParams::eSoftThresh*.
Segmentation was not done with percentages.

threshImage `const ccPelBuffer_const<c_UInt8>& threshImage() const;`

If you called **setSegmentationThreshImage()** to configure this **ccBlobParams**, this function returns the threshold image you specified.

Throws

ccBlobDefs::BadParams
Segmentation type is not *ccBlobParams::eThreshImage*.

scaleVal

```
c_UInt8 scaleVal() const;
```

If you called **setSegmentationNone()**, **setSegmentationMap** or **setSegmentationThreshImage()** to configure this **ccBlobParams**, this function returns the scaling value.

keepRLE

```
bool keepRLE() const;
void keepRLE(bool keep);
```

- `bool keepRLE() const;`

  Returns *true* if this **ccBlobParams** is configured to retain the **ccRLEBuffer** created when the input image is segmented.

- `void keepRLE(bool keep);`

  Configures this **ccBlobParams** to either retain or discard the **ccRLEBuffer** created when the input image is segmented.

  Parameters

keep	If *true*, this **ccBlobParams** will retain a copy of the **ccRLEBuffer**, if *false* the **ccRLEBuffer** will be discarded.

mask

```
const ccRLEBuffer& mask() const;
void mask(const ccRLEBuffer& m);
```

- `const ccRLEBuffer& mask() const;`

  Returns a reference to a **ccRLEBuffer** that contains the mask image that will be applied to the input image during segmentation.

  If no mask image has been specified for this **ccBlobParams**, calling **mask().isDegenerate()** returns *true*.

- `void mask(const ccRLEBuffer& theMask);`

  Specifies the mask to be applied to the input image before the image is segmented. Only pixels in the input image that correspond to nonzero pixels in *theMask* are included in the image to be segmented.

  Parameters

theMask	The mask image. *theMask* must have the same dimensions as the input image.

keepMasked

```
bool keepMasked() const;

void keepMasked(bool keep);
```

- `bool keepMasked() const;`

  Returns *true* if this **ccBlobParams** is configured to retain the **ccRLEBuffer** created by applying a mask image before the input image is segmented.

- `void keepMasked(bool keep);`

  Configures this **ccBlobParams** to either retain or discard the **ccRLEBuffer** image created by applying a mask image before the input image is segmented.

  Parameters

keep	If *true*, this **ccBlobParams** will retain a copy of the masked image, if *false* the masked image will be discarded.

morph

```
const cmStd vector<ceBlobParamsMorphOp>& morph() const;

void morph(const cmStd vector<ceBlobParamsMorphOp>& mOps);
```

- `const cmStd vector<ceBlobParamsMorphOp>& morph() const;`

  Returns a vector of **ccBlobParams::ceBlobParamsMophOp** that specifies the morphological operations that will be applied to the input image. Each element of the vector will have one of the following values:

  ccBlobParams::eMinH
  ccBlobParams::eMinV
  ccBlobParams::eMinS
  ccBlobParams::eMaxH
  ccBlobParams::eMaxV
  ccBlobParams::eMaxS
  ccBlobParams::eMinMaxH
  ccBlobParams::eMinMaxV
  ccBlobParams::eMinMaxS
  ccBlobParams::eMaxMinH
  ccBlobParams::eMaxMinV
  ccBlobParams::eMaxMinS

  If no morphological operations are specified, the returned vector has a length of 0. For more information, see the description of **ccBlobParams::ceBlobParamsMophOp**.

- `void morph(const cmStd vector<ceBlobParamsMorphOp>& mOps);`

  Specifies the morphological operations to be applied to the input image. You specify the morphological operations by supplying a vector of **ccBlobParams::ceBlobParamsMophOp**. Each element of the vector must have one of the following values:

  ccBlobParams::eMinH
  ccBlobParams::eMinV
  ccBlobParams::eMinS
  ccBlobParams::eMaxH
  ccBlobParams::eMaxV
  ccBlobParams::eMaxS
  ccBlobParams::eMinMaxH
  ccBlobParams::eMinMaxV
  ccBlobParams::eMinMaxS
  ccBlobParams::eMaxMinH
  ccBlobParams::eMaxMinV
  ccBlobParams::eMaxMinS

  To specify no morphological operations, supply a vector with a length of 0.

  Parameters

mOps	The morphological operations to perform.

keepMorphed

```
bool keepMorphed() const;
void keepMorphed(bool keep);
```

- `bool keepMorphed() const;`

  Returns *true* if this **ccBlobParams** is configured to retain the **ccRLEBuffer** created when the specified morphological operations are applied to the input image.

- `void keepMorphed(bool keep);`

  Configures this **ccBlobParams** to either retain or discard the **ccRLEBuffer** created when the specified morphological operations are applied to the input image.

  Parameters

keep	If *true*, this **ccBlobParams** will retain a copy of the image produced by the morphological operations, if *false* the image will be discarded.

connectivityType

```
ccBlobSceneDescription::Analysis connectivityType() const;

void connectivityType(
  ccBlobSceneDescription::Analysis cType);
```

- `ccBlobSceneDescription::Analysis connectivityType() const;`

  Returns the connectivity type configured for this **ccBlobParams**. The connectivity type is one of the following values:

  ccBlobSceneDescription::eGreyScale
  ccBlobSceneDescription::eLabelled
  ccBlobSceneDescription::eWholeImageGreyScale

- `void connectivityType(`
  `ccBlobSceneDescription::Analysis cType);`

  Sets the connectivity type for this **ccBlobParams**. The connectivity type must be one of the following values:

  ccBlobSceneDescription::eGreyScale
  ccBlobSceneDescription::eLabelled
  ccBlobSceneDescription::eWholeImageGreyScale

  Parameters

cType	The connectivity type to set.

connectivityCleanup

```
ccBlobSceneDescription::ConnectCleanup
  connectivityCleanup() const;

void connectivityCleanup(
  ccBlobSceneDescription::ConnectCleanup cType);
```

- `ccBlobSceneDescription::ConnectCleanup`
  `connectivityCleanup() const;`

  Returns the connectivity cleanup type configured for this **ccBlobParams**. The connectivity cleanup type is one of the following values:

  ccBlobSceneDescription::eNone
  ccBlobSceneDescription::ePrune
  ccBlobSceneDescription::eFill

- ```
  void connectivityCleanup(
    ccBlobSceneDescription::ConnectCleanup cType);
  ```

  Sets the connectivity cleanup type for this **ccBlobParams**. The connectivity cleanup type must be one of the following values:

  ccBlobSceneDescription::eNone
  ccBlobSceneDescription::ePrune
  ccBlobSceneDescription::eFill

  Parameters
  cType The connectivity cleanup type to set.

connectivityMinPels

```
c_Int32 connectivityMinPels() const;

void connectivityMinPels(c_Int32 min);
```

Specifies the minimum size, in pixels, of features returned by **cfBlobAnalysis()** when the **ccBlobSceneDescription::ConnectCleanup** value in effect is not *eNone.*

- ```
  c_Int32 connectivityMinPels() const;
  ```

  Returns the minimum feature size configured for this **ccBlobParams**.

- ```
  void connectivityMinPels(c_Int32 min);
  ```

  Sets the minimum feature size for this **ccBlobParams**.

  Parameters
  min The minimum feature size, in pixels.

  Throws
  ccBlobDefs::BadParams
  min is less than zero.

useMaskForInterior

```
bool useMaskForInterior() const;

void useMaskForInterior(bool useMaskForInterior);
```

This flag determines whether the image mask contained in this object will be used during the computation of **ccBlob::isInterior()**. If this flag is true and the mask is not degenerate, the mask is used. Otherwise, the mask is not used.
```

- `bool useMaskForInterior() const;`

  Returns the current *useMaskForInterior* flag, true or false.

- `void useMaskForInterior(bool useMaskForInterior);`

  Sets the *useMaskForInterior* flag, true or false.

  Parameters

  useMaskForInterior
  The new flag.

ccBlobResults

```
#include <ch_cvl/blobtool.h>

class ccBlobResults: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the results from a blob analysis.

Constructors/Destructors

ccBlobResults

```
ccBlobResults();

~ccBlobResults();
```

- `ccBlobResults();`

  Constructs a **ccBlobResults** with the following default values:

  - All timing information set to 0.
  - All intermediate images (RLE image, masked image, morphology output image) set to degenerate images.
  - Blob scene description set to *NULL*.

Operators

operator=

```
ccBlobResults& operator= (const ccBlobResults& rhs);
```

Assignment.

Public Member Functions

connectedBlobs

```
ccBlobSceneDescription * connectedBlobs() const;
```

Returns a pointer to the **ccBlobSceneDescription** produced by blob analysis.

Notes

The result structure retains ownership of the BSD object.

relinquishBSDOwnership

```
void relinquishBSDOwnership();
```

Sets this **ccBlobResults**'s internal **ccBlobSceneDescription** pointer to *NULL*. If you call this function, you *must* have already called **ccBlobSceneDescription::connectedBlobs()**, and you are responsible for deleting the **ccBlobSceneDescription** created by the call.

If you call this function without having called **ccBlobSceneDescription::connectedBlobs()**, you will not be able to obtain the results of the blob analysis, and your application will be unable to recover the memory allocated for the **ccBlobSceneDescription**.

RLETime

```
double RLETime() const;
```

Returns the amount of time, in seconds, required to encode the input image as a **ccRLEBuffer**.

rle

```
const ccRLEBuffer& rle() const;
```

Returns a **ccRLEBuffer** that contains the image produced during the segmentation step of this blob analysis.

maskTime

```
double maskTime() const;
```

Returns the amount of time, in seconds, required to apply the mask image to the input image.

masked

```
const ccRLEBuffer& masked() const;
```

Returns a **ccRLEBuffer** that contains the image produced by applying the mask image.

morphTime

```
double morphTime() const;
```

Returns the amount of time, in seconds, required to apply the specified morphological operations to the input image.

morphed

```
const ccRLEBuffer& morphed() const;
```

Returns a **ccRLEBuffer** that contains the image produced by applying the morphological operations.

connectTime `double connectTime() const;`

Returns the amount of time, in seconds, required to perform the connectivity analysis.

ccBlobResults

ccBlobSceneDescription

```
#include <ch_cvl/blobdesc.h>

class ccBlobSceneDescription: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains a **ccBlob** for each feature that was identified by the blob analysis.

Information about individual **ccBlob**s within a **ccBlobSceneDescription** can be obtained using the **ccBlob** member functions.

Constructors/Destructors

ccBlobSceneDescription

```
ccBlobSceneDescription();

~ccBlobSceneDescription();

ccBlobSceneDescription(const ccBlobSceneDescription& rhs);
```

- `ccBlobSceneDescription();`

  Constructs an empty **ccBlobSceneDescription**.

- `~ccBlobSceneDescription();`

  Destroys this **ccBlobSceneDescription** and all of its contained **ccBlob** objects.

Enumerations

Analysis

```
enum Analysis;
```

This enumeration defines the types of connectivity analysis that can be performed on an image.

Value	Meaning
eGreyScale	Connected object pixels are grouped and analyzed as individual features (blobs and holes). Blobs (made up of object pixels) are 8-connected. Holes (made up of background pixels) are 4-connected.
eLabelled	All pixels with the same label value are connected into individual features. No concept of object and background. All features are 6-connected.
eWholeImageGreyScale	All of the object pixels in the image are analyzed as a single object, regardless of whether or how they are connected to each other. No connectivity is performed.

ConnectCleanup

```
enum ConnectCleanup;
```

This enumeration defines the types of connectivity cleanup that can be performed during blob analysis.

Value	Meaning
eNone	No connectivity cleanup.

Value	Meaning
ePrune	Features below a specified size threshold are not counted as part of the topology of the blob scene. The size and shape measures of features that contain pruned features are reported taking into account the area of the pruned features.
eFill	Features below a specified size threshold are filled in with the pixel value of the surrounding feature.

SortOrder

```
enum SortOrder;
```

This enumeration defines the available sort orders. You can specify the sort order and measure that determines the order in which features are returned to you.

Value	Meaning
eDescending	Features are returned from largest to smallest value.
eAscending	Features are returned from smallest to largest value.

Public Member Functions

isDegenerate

```
bool isDegenerate() const;
```

Returns true if this **ccBlobSceneDescription** was default constructed.

connectivityKind

```
Analysis connectivityKind() const;
```

Returns the type of connectivity analysis used to generate the information in this **ccBlobSceneDescription**. Returns one of the following values:

ccBlobSceneDescription::eGreyScale
ccBlobSceneDescription::eLabelled
ccBlobSceneDescription::eWholeImageGreyScale

scaleVal

```
c_UInt8 scaleVal() const;
```

Returns the scaling value used to perform the grey-scale connectivity analysis that produced this **ccBlobSceneDescription**.

cleanupKind

```
ConnectCleanup cleanupKind() const;
```

Returns the type of connectivity cleanup performed to generate the information in this **ccBlobSceneDescription**. Returns one of the following values:

ccBlobSceneDescription::eNone
ccBlobSceneDescription::ePrune
ccBlobSceneDescription::eFill

minPels

```
c_Int32 minPels() const;
```

Returns the minimum feature size applied during the connectivity cleanup operation performed to generate the information in this **ccBlobSceneDescription**.

sceneWindow

```
ccPelRect sceneWindow() const;
```

Returns the dimensions of the **ccRLEBuffer** upon which connectivity analysis was performed.

clientFromImageXformBase

```
cc2XformBasePtrh_const clientFromImageXformBase() const;

void clientFromImageXformBase(
  const cc2XformBasePtrh_const& xform, bool copy = true);

void clientFromImageXformBase(const cc2XformBase& xform);
```

- `cc2XformBasePtrh_const clientFromImageXformBase() const;`

  Returns the transformation object that it is suitable for transforming the **ccRLEBuffer** coordinates to the client's reference coordinates.

- ```
  void clientFromImageXformBase(
    const cc2XformBasePtrh_const& xform, bool copy = true);
  ```

  Sets this **ccBlobSceneDescription**'s transformation object suitable for transforming the **ccRLEBuffer** coordinates to the client's reference coordinates.
  ```

Parameters

xform	A pointer handle to the transformation object.
copy	If true, this window is assigned a copy of *xform*. If false, this window is assigned a reference to *xform*.

Notes

If *copy* is false, you must ensure that *xform* will not change during this **ccBlobSceneDescription**'s lifetime.

- `void clientFromImageXformBase(const cc2XformBase& xform);`

Sets this window's transformation object suitable for transforming the **ccRLEBuffer** coordinates to the client's reference coordinates.

Parameters

xform	A transformation object.

Notes

This setter always makes a copy of *xform*.

Both **imageFromClientXformBase()** and **clientFromImageXformBase()** setters set the same object. Use one or the other, but not both. The deprecated functions **imageFromClientXform()** and **clientFromImageXform()** setters also operate on the same object.

imageFromClientXformBase

```
cc2XformBasePtrh_const imageFromClientXformBase() const;

void imageFromClientXformBase(
  const cc2XformBasePtrh_const& xform, bool copy = true);

void imageFromClientXformBase(const cc2XformBase& xform);
```

- `cc2XformBasePtrh_const imageFromClientXformBase() const;`

Returns the transformation object suitable for transforming the client's reference coordinates to the **ccRLEBuffer**'s coordinates.

- `void imageFromClientXformBase(`
  `const cc2XformBasePtrh_const& xform, bool copy = true);`

Sets this **ccBlobSceneDescription**'s transformation object suitable for transforming the client's reference coordinates to the **ccRLEBuffer**'s coordinates.

Parameters

xform	A pointer handle to the transformation object.
copy	If true, this window is assigned a copy of *xform*. If false, this window is assigned a reference to *xform*.

Notes

If *copy* is false, you must ensure that *xform* will not change during this window's lifetime.

- `void imageFromClientXformBase(const cc2XformBase& xform);`

Sets this **ccBlobSceneDescription**'s transformation object suitable for transforming the client's reference coordinates to the **ccRLEBuffer**'s coordinates.

Parameters

xform	A transformation object.

Notes

This setter always makes a copy of *xform*.

Both **imageFromClientXformBase()** and **clientFromImageXformBase()** setters set the same object. Use one or the other, but not both. The deprecated functions **imageFromClientXform()** and **clientFromImageXform()** setters also operate on the same object.

imageFromClientXform

```
cc2Xform imageFromClientXform() const;

void imageFromClientXform(
  const cc2Xform& imageFromClient);
```

- `cc2Xform imageFromClientXform() const;`

Returns a **cc2Xform** that describes the transformation between the client coordinate system associated with the **ccRLEBuffer** upon which connectivity analysis was performed and its image coordinate system.

Notes

This function is deprecated; use **imageFromClientXformBase()** instead.

- ```
  void imageFromClientXform(
    const cc2Xform& imageFromClient);
  ```

  Sets the **cc2Xform** that describes the transformation between the client coordinate system associated with the **ccRLEBuffer** upon which connectivity analysis was performed and its image coordinate system.

  Setting this **cc2Xform** also sets the **cc2Xform** that describes the transformation between image coordinates and client coordinates.

  Parameters
  imageFromClient The transformation to set.

  Notes
  This function is deprecated; use **imageFromClientXformBase()** instead.

clientFromImageXform

```
cc2Xform clientFromImageXform() const;

void clientFromImageXform(
  const cc2Xform& clientFromImage);
```

- ```
  cc2Xform clientFromImageXform() const;
  ```

  Returns a **cc2Xform** that describes the transformation between the image coordinate system associated with the **ccRLEBuffer** upon which connectivity analysis was performed and its client coordinate system.

  Notes
  This function is deprecated; use **clientFromImageXformBase()** instead.

- ```
  void clientFromImageXform(
    const cc2Xform& clientFromImage);
  ```

  Sets the **cc2Xform** that describes the transformation between the image coordinate system associated with the **ccRLEBuffer** upon which connectivity analysis was performed and its client coordinate system.

  Setting this **cc2Xform** also sets the **cc2Xform** that describes the transformation between client coordinates and image coordinates.

  Parameters
  clientFromImage The transformation to set.

  Notes
  This function is deprecated; use **clientFromImageXformBase()** instead.
  ```

numBlobs

```
c_Int32 numBlobs(bool filtered = true) const;
```

Returns the number of features in this **ccBlobSceneDescription**. In the case of grey-scale connectivity, this includes both blobs and holes.

Parameters

filtered — If *filtered* is true, only features that meet the current filtering criteria are included in the returned count.

getBlob

```
const ccBlob* getBlob(c_Int32 id) const;
```

Returns a pointer to the **ccBlob** that has the specified ID.

Parameters

id — The ID of the **ccBlob** to return. *id* must be between 1 and the number of features in this **ccBlobSceneDescription**, inclusive.

preCompute

```
void preCompute(ccBlob::Measure theMeasure,
  bool filtered = true) const;
```

Pre-computes the specified measure for all features in this **ccBlobSceneDescription**.

Ordinarily, a particular measure for a particular feature is not computed until you specifically request it, or until the measure is used as a sorting or filtering criterion. You call this function to force the computation of a particular measure for all of the features in this **ccBlobSceneDescription**.

Parameters

theMeasure — The blob measure to pre-compute. *theMeasure* must be one of the following values:

ccBlob::eLabel
ccBlob::eArea
ccBlob::eNumSteps
ccBlob::ePerimeter
ccBlob::eNumChildren
ccBlob::eCenterMassX
ccBlob::eCenterMassY
ccBlob::eInertiaX
ccBlob::eInertiaY
ccBlob::eInertiaMin
ccBlob::eInertiaMax
ccBlob::eElongation
ccBlob::eAngle
ccBlob::eAcircularity
ccBlob::eAcircularityRms
ccBlob::eIsInterior

ccBlob::eImageBoundCenterX
ccBlob::eImageBoundCenterY
ccBlob::eImageBoundMinX
ccBlob::eImageBoundMaxX
ccBlob::eImageBoundMinY
ccBlob::eImageBoundMaxY
ccBlob::eImageBoundWidth
ccBlob::eImageBoundHeight
ccBlob::eImageBoundAspect
ccBlob::eMedianX
ccBlob::eMedianY
ccBlob::eBoundCenterX
ccBlob::eBoundCenterY
ccBlob::eBoundMinX
ccBlob::eBoundMaxX
ccBlob::eBoundMinY
ccBlob::eBoundMaxY
ccBlob::eBoundWidth
ccBlob::eBoundHeight
ccBlob::eBoundAspect
ccBlob::eBoundPrincipalMinX
ccBlob::eBoundPrincipalMaxX
ccBlob::eBoundPrincipalMinY
ccBlob::eBoundPrincipalMaxY
ccBlob::eBoundPrincipalWidth
ccBlob::eBoundPrincipalHeight
ccBlob::eBoundPrincipalAspect

filtered — If *filtered* is true, then *theMeasure* is pre-computed only for features that meet the current filtering criteria.

setFilter

```
void setFilter(ccBlob::Measure theMeasure, double loLimit,
  double hiLimit, bool invert=false);
```

Sets a filtering criterion for the specified measure. Only features which meet the specified criterion will be considered and/or returned by certain **ccBlob** and **ccBlobSceneDescription** member functions that, optionally, take filtering into account.

You can specify only one filter per measure. If you specify a filter for a measure which already has a filter, the error *ccBlobSceneDescription:DuplicateFilter* is thrown.

You can specify any number of filters for a single **ccBlobSceneDescription**. Each filter you specify is assigned an index value, starting with 0.

Parameters

theMeasure The measure on which to apply a filter. *theMeasure* must be one of the following values:

ccBlob::eLabel
ccBlob::eArea
ccBlob::eNumSteps
ccBlob::ePerimeter
ccBlob::eNumChildren
ccBlob::eCenterMassX
ccBlob::eCenterMassY
ccBlob::eInertiaX
ccBlob::eInertiaY
ccBlob::eInertiaMin
ccBlob::eInertiaMax
ccBlob::eElongation
ccBlob::eAngle
ccBlob::eAcircularity
ccBlob::eAcircularityRms
ccBlob::eIsInterior
ccBlob::eImageBoundCenterX
ccBlob::eImageBoundCenterY
ccBlob::eImageBoundMinX
ccBlob::eImageBoundMaxX
ccBlob::eImageBoundMinY
ccBlob::eImageBoundMaxY
ccBlob::eImageBoundWidth
ccBlob::eImageBoundHeight
ccBlob::eImageBoundAspect
ccBlob::eMedianX
ccBlob::eMedianY
ccBlob::eBoundCenterX
ccBlob::eBoundCenterY
ccBlob::eBoundMinX
ccBlob::eBoundMaxX
ccBlob::eBoundMinY
ccBlob::eBoundMaxY
ccBlob::eBoundWidth
ccBlob::eBoundHeight
ccBlob::eBoundAspect
ccBlob::eBoundPrincipalMinX
ccBlob::eBoundPrincipalMaxX
ccBlob::eBoundPrincipalMinY
ccBlob::eBoundPrincipalMaxY

ccBlob::eBoundPrincipalWidth
ccBlob::eBoundPrincipalHeight
ccBlob::eBoundPrincipalAspect

loLimit — The low limit for the filter. Only features with values for the specified measure that are greater than or equal to *loLimit* are returned.

hiLimit — The high limit for the filter. Only features with values for the specified measure that are less than or equal to *hiLimit* are returned.

invert — If *invert* is *false* (the default), then only features where the value of *theMeasure* is between *loLimit* and *hiLimit* are returned. If *invert* is *true*, then only features where the value of *theMeasure* is less than *loLimit* or greater than *hiLimit* are returned.

Throws

ccBlobSceneDescription::DuplicateFilter
You have specified a measure for which there already is a filter.

getFilter

```
bool getFilter(ccBlob::Measure theMeasure, double& loLimit,
  double& hiLimit, bool& invert) const;

void getFilter(c_Int32 index, ccBlob::Measure& theMeasure,
  double& loLimit, double& hiLimit, bool& invert) const;
```

- `bool getFilter(ccBlob::Measure theMeasure, double& loLimit, double& hiLimit, bool& invert) const;`

Gets the currently specified filtering criteria for the specified measure.

Parameters

theMeasure — The measure about which to obtain filter information. *theMeasure* must be one of the following values:

ccBlob::eLabel
ccBlob::eArea
ccBlob::eNumSteps
ccBlob::ePerimeter
ccBlob::eNumChildren
ccBlob::eCenterMassX
ccBlob::eCenterMassY
ccBlob::eInertiaX
ccBlob::eInertiaY
ccBlob::eInertiaMin
ccBlob::eInertiaMax

ccBlob::eElongation
ccBlob::eAngle
ccBlob::eAcircularity
ccBlob::eAcircularityRms
ccBlob::eIsInterior
ccBlob::eImageBoundCenterX
ccBlob::eImageBoundCenterY
ccBlob::eImageBoundMinX
ccBlob::eImageBoundMaxX
ccBlob::eImageBoundMinY
ccBlob::eImageBoundMaxY
ccBlob::eImageBoundWidth
ccBlob::eImageBoundHeight
ccBlob::eImageBoundAspect
ccBlob::eMedianX
ccBlob::eMedianY
ccBlob::eBoundCenterX
ccBlob::eBoundCenterY
ccBlob::eBoundMinX
ccBlob::eBoundMaxX
ccBlob::eBoundMinY
ccBlob::eBoundMaxY
ccBlob::eBoundWidth
ccBlob::eBoundHeight
ccBlob::eBoundAspect
ccBlob::eBoundPrincipalMinX
ccBlob::eBoundPrincipalMaxX
ccBlob::eBoundPrincipalMinY
ccBlob::eBoundPrincipalMaxY
ccBlob::eBoundPrincipalWidth
ccBlob::eBoundPrincipalHeight
ccBlob::eBoundPrincipalAspect

loLimit — A reference to a **double** into which the low limit for the filter is written.

hiLimit — A reference to a **double** into which the high limit for the filter is written.

invert — A **bool** into which the value of the invert flag is written.

- `void getFilter(c_Int32 index, ccBlob::Measure& theMeasure, double& loLimit, double& hiLimit, bool& invert) const;`

  Gets the filtering criteria for the specified filter. As you apply filters to a **ccBlobSceneDescription**, the filters are assigned an index value, starting with 0 for the first filter you apply. This function lets you obtain information about a filter based on its index value.

  Parameters

index	The index of the filter. This value must be greater than or equal to 0 and less than or equal to the number of filters.
theMeasure	A reference to a **ccBlob::Measure** into which the measure associated with this filter is written.
loLimit	A reference to a **double** into which the low limit for the filter is written.
hiLimit	A reference to a **double** into which the high limit for the filter is written.
invert	A **bool** into which the value of the invert flag is written.

unSetFilter `void unSetFilter(ccBlob::Measure theMeasure);`

Clears the filter associated with the specified measure. The indices of other filters may change as a result of calling this function.

Parameters

theMeasure	The measure to clear. *theMeasure* must be one of the following values: *ccBlob::eLabel* *ccBlob::eArea* *ccBlob::eNumSteps* *ccBlob::ePerimeter* *ccBlob::eNumChildren* *ccBlob::eCenterMassX* *ccBlob::eCenterMassY* *ccBlob::eInertiaX* *ccBlob::eInertiaY* *ccBlob::eInertiaMin* *ccBlob::eInertiaMax* *ccBlob::eElongation* *ccBlob::eAngle* *ccBlob::eAcircularity* *ccBlob::eAcircularityRms*

ccBlob::eIsInterior
ccBlob::eImageBoundCenterX
ccBlob::eImageBoundCenterY
ccBlob::eImageBoundMinX
ccBlob::eImageBoundMaxX
ccBlob::eImageBoundMinY
ccBlob::eImageBoundMaxY
ccBlob::eImageBoundWidth
ccBlob::eImageBoundHeight
ccBlob::eImageBoundAspect
ccBlob::eMedianX
ccBlob::eMedianY
ccBlob::eBoundCenterX
ccBlob::eBoundCenterY
ccBlob::eBoundMinX
ccBlob::eBoundMaxX
ccBlob::eBoundMinY
ccBlob::eBoundMaxY
ccBlob::eBoundWidth
ccBlob::eBoundHeight
ccBlob::eBoundAspect
ccBlob::eBoundPrincipalMinX
ccBlob::eBoundPrincipalMaxX
ccBlob::eBoundPrincipalMinY
ccBlob::eBoundPrincipalMaxY
ccBlob::eBoundPrincipalWidth
ccBlob::eBoundPrincipalHeight
ccBlob::eBoundPrincipalAspect

clearFilters

```
void clearFilters();
```

Clears all filters associated with this **ccBlobSceneDescription**.

numFilters

```
c_Int32 numFilters() const;
```

Returns the number of filters applied to this **ccBlobSceneDescription**.

setSort

```
void setSort(ccBlob::Measure theMeasure,
  SortOrder theOrder);
```

Sets the sort order for this **ccBlobSceneDescription**. The sort order controls the order in which individual **ccBlob**s are returned.

Parameters

theMeasure The measure upon which to sort the features. *theMeasure* must be one of the following values:

ccBlob::eLabel
ccBlob::eArea
ccBlob::eNumSteps
ccBlob::ePerimeter
ccBlob::eNumChildren
ccBlob::eCenterMassX
ccBlob::eCenterMassY
ccBlob::eInertiaX
ccBlob::eInertiaY
ccBlob::eInertiaMin
ccBlob::eInertiaMax
ccBlob::eElongation
ccBlob::eAngle
ccBlob::eAcircularity
ccBlob::eAcircularityRms
ccBlob::eIsInterior
ccBlob::eImageBoundCenterX
ccBlob::eImageBoundCenterY
ccBlob::eImageBoundMinX
ccBlob::eImageBoundMaxX
ccBlob::eImageBoundMinY
ccBlob::eImageBoundMaxY
ccBlob::eImageBoundWidth
ccBlob::eImageBoundHeight
ccBlob::eImageBoundAspect
ccBlob::eMedianX
ccBlob::eMedianY
ccBlob::eBoundCenterX
ccBlob::eBoundCenterY
ccBlob::eBoundMinX
ccBlob::eBoundMaxX
ccBlob::eBoundMinY
ccBlob::eBoundMaxY
ccBlob::eBoundWidth
ccBlob::eBoundHeight
ccBlob::eBoundAspect
ccBlob::eBoundPrincipalMinX
ccBlob::eBoundPrincipalMaxX
ccBlob::eBoundPrincipalMinY
ccBlob::eBoundPrincipalMaxY

ccBlob::eBoundPrincipalWidth
ccBlob::eBoundPrincipalHeight
ccBlob::eBoundPrincipalAspect

theOrder — The order in which to sort the features. *theOrder* must be one of the following values:

ccBlobSceneDescription::eAscending
ccBlobSceneDescription::eDescending

getSort

```
bool getSort(ccBlob::Measure& theMeasure,
  SortOrder& theOrder) const;
```

Returns *true* if a call to **setSort()** has been made to enable sorting for this **ccBlobSceneDescription**. Otherwise the function returns *false*. If the function returns *true*, the current sort order and sort measure are placed in *theOrder* and *theMeasure*.

Parameters

theMeasure — A reference to a **ccBlob::Measure** into which is written the measure upon which the features in this **ccBlobSceneDescription** are sorted.

theOrder — A reference to a **ccBlobSceneDescription::SortOrder** into which the sort order is written.

disableSort

```
void disableSort();
```

Disables sorting for this **ccBlobSceneDescription**. If sorting is disabled, then features are returned in an unspecified order and **getSort()** returns *false*.

Calling this function is the *only* way to prevent the Blob tool from sorting features. If you have never made a call to **setSort()**, features are sorted by area. If you call **clearSort()**, features will be sorted using the measure supplied to the most recent call to **setSort()**, or by area if no call to **setSort()** has been made.

Calling this function if sorting is already disabled has no effect.

clearSort

```
void clearSort();
```

Calling this function causes **getSort()** to return false, but it does not actually disable the sorting of features. To disable the sorting of features, use **disableSort()**.

If you call this function, features will still be sorted using the measure supplied to the most recent call to **setSort()**. If no calls have been made to **setSort()**, then the features are sorted by area.

This function is present for backward compatibility only.

firstTop

```
const ccBlob* firstTop(bool filtered = true) const;
```

Returns a pointer to a **ccBlob** representing the first top-level feature in this **ccBlobSceneDescription**. You can specify that only features meeting the current filtering criteria be returned by this function.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

lastTop

```
const ccBlob* lastTop(bool filtered = true) const;
```

Returns a pointer to a **ccBlob** representing the last top-level feature in this **ccBlobSceneDescription**. You can specify that only features meeting the current filtering criteria be returned by this function.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

allBlobs

```
cmStd vector<const ccBlob*> allBlobs(bool filtered = true)
  const;
```

Returns a vector of pointers to **ccBlob**. The vector contains pointers to all the features in this **ccBlobSceneDescription**, in the current sort order. You can specify that only features meeting the current filtering criteria be returned by this function.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

Notes

There is no relationship between a feature's ID and its position in the vector returned by **allBlobs()**.

makeImage

```
ccPelBuffer<c_UInt8> makeImage(bool filtered = true,
  c_UInt8 background=0) const;
```

Returns a **ccPelBuffer<c_UInt8>** that contains an image of this **ccBlobSceneDescription**.

Parameters

filtered — If *true*, only features that meet the current filtering criteria are returned, and features are sorted according to the current sort criteria. If *false*, no filtering or sorting is performed.

background — Specifies the pixel value for the background of the returned image.

extremaExcludeArea

```
void extremaExcludeArea(double clientArea);

double extremaExcludeArea(bool& isClient, bool& isPercent)
  const;
```

- `void extremaExcludeArea(double clientArea);`

  Sets the amount of each feature to exclude when computing measures related to the arbitrarily aligned bounding box.

  Parameters

  clientArea — The amount of the feature to exclude in terms of client coordinate system-sized pixels.

- `double extremaExcludeArea(bool& isClient, bool& isPercent) const;`

  Returns the amount of each feature that this **ccBlobSceneDescription** is currently configured to exclude when computing the arbitrarily aligned bounding box.

  Parameters

  isClient — A reference to a **bool** into which *true* is written if the returned value is given in client coordinate-sized pixels. If *isClient* is set to *false*, then the return value is given in image coordinate-sized pixels.

  isPercent — A reference to a **bool** into which *true* is written if the returned value is given in as a percentage of the area of the feature. If *isPercent* is set to *false*, the return value is given in terms of pixel area.

extremaExcludeAreaPels

```
void extremaExcludeAreaPels(double pelArea);
```

Sets the amount of each feature to exclude when computing the arbitrarily aligned bounding box.

Parameters

clientArea The amount of the feature to exclude in terms of image coordinate system-sized pixels.

extremaExcludeAreaPercent

```
void extremaExcludeAreaPercent(double percent);
```

Sets the amount of each feature to exclude when computing the arbitrarily aligned bounding box.

Parameters

clientArea The amount of the feature to exclude in terms of the percentage of pixels to exclude.

extremaAngle

```
ccRadian extremaAngle() const;

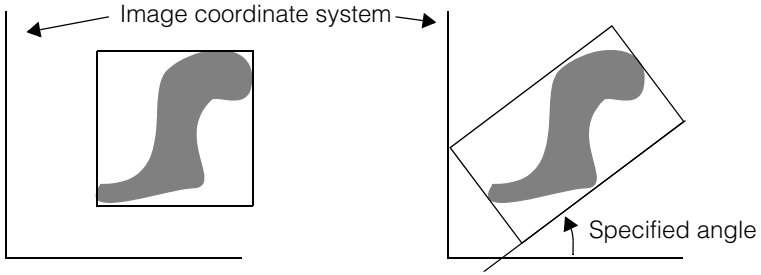
void extremaAngle(const ccRadian& angle);
```

- `ccRadian extremaAngle() const;`

  Returns the angle at which various measures are computed, relative to the client coordinate system's x-axis, in radians, for all features in this **ccBlobSceneDescription**.

- `void extremaAngle(const ccRadian& angle);`

  Sets the angle at which the bounding box is computed, relative to the client coordinate system's x-axis, in radians, for all features in this **ccBlobSceneDescription**. The following diagram illustrates the effect of specifying an angle value:



No angle specified Angle specified

Parameters

angle — The angle at which to compute bounding boxes and bounding box-related measures for features in this **ccBlobSceneDescription**.

draw

```
void draw(ccGraphicList& graphList,
 c_UInt32 drawMode = ccBlobDefs::eDrawStandard,
 bool filtered=true) const;
```

Appends result graphics for all of the features in this **ccBlobSceneDescription** to the supplied **ccGraphicList**. To append result graphics for only those features that meet the current filtering criteria, specify true for the *filtered* argument.

The graphics are drawn in client coordinates.

Parameters

graphList — The graphics list to append the graphics to.

drawMode — The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccBlobDefs::eDrawImageCenterOfMass
ccBlobDefs::eDrawCenterOfMass
ccBlobDefs::eDrawLabel
ccBlobDefs::eDrawImageBoundingBox
ccBlobDefs::eDrawBoundingBox
ccBlobDefs::eDrawBoundingTrace
ccBlobDefs::eDrawStandard

filtered — If true, only features which meet the current filtering criteria have results drawn.

ccBoard

```
#include <ch_cvl/board.h>

class ccBoard;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This abstract class is used to describe Cognex hardware such as frame grabbers. For each physical Cognex vision device (frame grabber) installed on your PC, there is a single object derived from **ccBoard** that refers to it.

For cameras that do not use frame grabbers, such as GigE Vision cameras, there is a single **ccBoard** object that represents all cameras of the same type.

Your application can get a **ccBoard** object by using the **get()** function.

Constructors/Destructors

Construction and destruction of derived classes is done automatically.

Enumerations

ceState

```
enum ceState;
```

This enumeration defines the current state of the hardware.

Value	Meaning
ckNotSupported = 0	Not supported for the board.
ckReal = 1	Board is physically connected.
ckVirtual = 2	Board is not connected/missing, for example, a GigE camera that lost power.
ckMisconfigured = 3	Board is on a different subnet from the NIC.

Public Member Functions

getHardwareStatus

```
virtual ceState getHardwareStatus(bool cached);
```

Gets the current state of the hardware. See `enum ceState` for details.

Notes

This should be used only by **ccBoard** objects that support dynamic refresh.

name

```
const TCHAR* name();
```

Return the name of this board as a null-terminated string.

serialNumber

```
ccCvlString serialNumber() const;
```

Returns an ASCII string containing the serial number of the Cognex hardware device represented by this **ccBoard**.

Throws

ccBoard::BadEERAMContents
The board serial number cannot be read because the EERAM contents are invalid.

upgradeCode

```
virtual ccCvlString upgradeCode() const;
```

Returns the upgrade code for this board.

Notes

If this board does not have an upgrade code, this function returns a zero-length string.

Throws

ccBoard::BadEERAMContents
The EERAM cannot be read.

writeEERAMData

```
void writeEERAMData(c_Int32 offset,
 const cmStd vector<c_UInt8> &data);
```

Write user-defined data into EERAM starting at the specified offset.

Parameters

offset — The location within user EERAM area at which to start writing data. Must be in the range 0 through **sizeEERAMData()**.

data — The data to write.

Throws

ccBoard::BadParams

data is too big to fit into user EERAM area.

readEERAMData

```
void readEERAMData(c_Int32 offset,
  cmStd vector<c_UInt8> &data, c_Int32 nBytes);
```

Read data from the user EERAM area.

Parameters

offset	The location within user EERAM area at which to start reading data. Must be in the range 0 through **sizeEERAMData()**.
data	The data read from user EERAM.
nBytes	The number of bytes to read.

Throws

ccBoard::BadParams

nBytes is too big and would read beyond the end of user EERAM.

sizeEERAMData

```
c_Int32 sizeEERAMData() const;
```

Return the size of the user EERAM data area.

Notes

Not all hardware devices provide a user EERAM area. If a device does not support a user EERAM data area, this function returns zero.

Static Functions

count

```
static c_Int32 count();
```

Returns the number of Cognex boards installed.

Notes

This function is not available at static initialization time.

For cameras that appear as boards to CVL (GigE Vision cameras, for example, the count of the number of cameras is available only after the cameras have finished their network negotiation. Use **isEnumerationComplete()** to determine when this is the case.

This function is intended for host use. If you call **count()** from an application running on an embedded system, such as the Cognex MVS-82400, it always returns 1 no matter how many boards are installed in the host system.

Throws

ccBoard::StaticInitDisorder
This function was called during static initialization time.

get

```
static ccBoard& get(c_Int32 i = 0);
```

Return the *i*th board installed. Board numbering starts at 0.

Parameters

i The board number.

Throws

ccBoard::BadParams
i is not within the range of 0 through **count()** - 1, or no boards are installed (**count()** returns 0).

ccBoard::StaticInitDisorder
This function was called during static initialization time.

Notes

This function is not available at static initialization time.

Boards are sorted alphabetically in increasing order according to board type. Within a given board type, the order may be operating-system and CVL-version dependent.

isEnumerationComplete

```
static bool isEnumerationComplete(double limitSec=0.0);
```

Returns True if the enumeration of boards is complete.

Parameters

limitSec The number of seconds to wait before returning if enumeration is not complete.

RefreshCameraList

```
static void RefreshCameraList();
```

Refreshes the list of hardware by detecting newly added devices.

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding

The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse

The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error, a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8100m**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occurred while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

ccBoundary

```
#include <ch_cvl/pmibnd.h>

class ccBoundary;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains boundary data. The member functions of the class return information about the geometrical properties of the boundary and allow for geometrical comparisons between two boundaries. This **ccBoundary** can be the trained, matched, extra or missing boundary returned by PatInspect. The boundary information embodied by the class includes:

- The position of each boundary point
- The direction of each boundary point
- The weight of each boundary point
- The matchQuality value of each boundary point
- The minimum enclosing rectangle of the boundary
- The mean position of the boundary
- The weighted mean position of the boundary.

Constructors/Destructors

ccBoundary

```
ccBoundary();

ccBoundary(const cmStd vector<ccInspectBPoint>& points,
  ccBoundaryDefs::BoundaryType type=
                                    ccBoundaryDefs::eTrained,
  const cc2Xform &pose=cc2Xform::I);

ccBoundary(const cmStd vector<cc2Vect>& points,
  const cc2Xform &pose=cc2Xform::I);
```

- `ccBoundary();`

  Creates a **ccBoundary** that has no points.

- ```
  ccBoundary(const cmStd vector<ccInspectBPoint>& points,
    ccBoundaryDefs::BoundaryType type=
                                        ccBoundaryDefs::eTrained,
    const cc2Xform &pose=cc2Xform::I);
  ```

  Creates a **ccBoundary** from the boundary type specified by *type*.

  Parameters

points	The list of boundary points of this **ccBoundary**.
type	The type of boundary. *type* must be one of the following: *ccBoundaryDefs::eTrainedt* *ccBoundaryDefs::eMatch* *ccBoundaryDefs::eMissing* *ccBoundaryDefs::eExtra*
pose	The **cc2Xform** that maps the boundary to client coordinates.

  Notes

  If type is *ccBoundaryDefs::eTrained* only the boundary points from the trained pattern that are matched at run-time are included in this **ccBoundary**.

- ```
  ccBoundary(const cmStd vector<cc2Vect>& points,
    const cc2Xform &pose=cc2Xform::I);
  ```

  Creates a **ccBoundary** from the *points* list. The direction associated with each boundary point is computed from the supplied list of points. The weight of each boundary point is set to 1.0. The matchQuality value of each boundary point is set to 0.0.

  Parameters

points	The list of points that make up the boundary.
pose	The **cc2Xform** that maps *points* to client coordinates.

  Notes

  This constructor is provided for testing purposes.

Public Member Functions

unweightedMeanPosition

```
cc2Vect unweightedMeanPosition() const;
```

Returns the unweighted mean position of this **ccBoundary**.

Notes

The unweighted mean position may not lie on the boundary. The unweighted mean position is computed by the constructor.

weightedMeanPosition

```
cc2Vect weightedMeanPosition() const;
```

Returns the weighted mean position of this **ccBoundary**.

Notes

The weighted mean position may not lie on the boundary. The weighted mean position is computed by the constructor.

area

```
double area(
  ccBoundaryDefs::AreaMethod method=
  ccBoundaryDefs::eSimple)const;
```

Returns the area of this **ccBoundary** according to the modality specified by *method*. If *method* is ***ccBoundaryDefs::eSimple*** the function computes the area of the polygon resulting from connecting adjacent boundary points with straight segments (if the boundary is open the two end-points are connected with a straight segment to close the polygon). If *method* is ***ccBoundaryDefs::eConvex*** the function computes the area of the convex hull of this **ccBoundary** (the convex hull of a boundary is the smallest enclosing polygon of the boundary, see *ccBoundaryDefs*).

Parameters

method — The modality used to compute the area of this **ccBoundary**. *method* must be one of the following:
ccBoundaryDefs::eSimplet
ccBoundaryDefs::eConvex

Throws

ccBoundaryDefs::DegenerateShape
method is ***ccBoundaryDefs::eSimple*** and the shape of the polygon that connects the boundary points is not simple (this means that the straight line that closes the boundary intersect the boundary).

Notes

The area cannot be computed if this **ccBoundary** crosses on itself.

convexHull

```
ccBoundary convexHull() const;
```

Returns the boundary that is the convex hull of this **ccBoundary**. The convex hull is the minimum enclosing polygon of this ccBoundary.

rect

```
ccRect rect() const;
```

Returns the minimum enclosing rectangle of this **ccBoundary** in client coordinates. The minimum enclosing rectangle of this **ccBoundary** is computed by the constructor.

draw

```
void draw(ccUITablet &tablet,
  bool connected=true,
  const ccColor &color=ccColor::greenColor(),
  const ccUITablet::Layers &layer=
  ccUITablet::eImageLayer) const;
```

Draws this **ccBoundary** in the supplied **ccUITablet** *tablet*.

Parameters

tablet	The tablet used to draw this **ccBoundary**.
connected	If true the function draws the polygon obtained by connecting the adjacent boundary points of this **ccBoundary** with straight segments.
color	The color of this **ccBoundary**.
layer	The layer into which this **ccBoundary** is drawn.

minDist

```
double minDist(const ccBoundary &bnd2,
  cc2Vect &pnt1,cc2Vect &pnt2,
  ccBoundaryDefs::DistanceMethod method=
  ccBoundaryDefs::eExhaustive) const;
```

Returns the distance between this **ccBoundary** and the boundary *bnd2* according to the modality specified by *method*. The positions of the two closest points are stored in *pnt1* (for this **ccBoundary**) and *pnt2* (for *bnd2*). The two closest points depend on the modality chosen to measure the distance between the two boundaries (see *ccBoundaryDefs*).

Parameters

method	The mode used to compute the distance between boundaries. *method* must be one of the following: *ccBoundaryDefs::eExhaustive* *ccBoundaryDefs::eMinEndPnt* *ccBoundaryDefs::eMeanPosition*
pnt1	The closest boundary point of this **ccBoundary** to *bnd2*.
pnt2	The closest boundary point of *bnd2* to this **ccBoundary**.

Throws

ccBoundaryDefs::DegenerateShape	This **ccBoundary** or *bnd2* or both have no points.

length

```
double length(bool assumeClosed=false) const;
```

Returns the length of this **ccBoundary** in client coordinates.

Parameters

assumeClosed If true the function returns the length of the closed shape obtained by joining the first and last point of this **ccBoundary** with a straight line.

Notes

The length returned is not the number of points in this **ccBoundary**.

inside

```
bool inside(const cc2Vect &pnt1) const;
```

Returns true if the point *pnt1* is inside this **ccBoundary**. Returns false if *pnt1* is outside or exactly on this **ccBoundary**.

Parameters

pnt1 The point whose position relative to this **ccBoundary** is evaluated.

angleBetweenSegments

```
ccRadian angleBetweenSegments(const ccBoundary &bnd2)
 const;
```

Returns the angle between this **ccBoundary** and *bnd2*. The angle between the two boundaries is defined as the angle between the best fitting line of this **ccBoundary** and the best fitting line of *bnd2*. The angle returned is from 0 through 2π radians.

Parameters

bnd2 The boundary whose best fitting line is used together with the best fitting line of this **ccBoundary** to compute the angle between the two boundaries.

Throws

ccBoundaryDefs::InvalidBoundaryCondition
The line fit to this **ccBoundary** or *bnd2* fails.

ccBoundaryDefs::DegenerateShape
This **ccBoundary** or *bnd2* or both have only one point.

measureDifferences

```
cmStd vector <cc2Vect> measureDifferences(
 const ccBoundary &bnd,
 ccBoundaryDefs::MeasurementMethod method =
```

```
ccBoundaryDefs::eParallel,
ccBoundaryDefs::MeasurementAccuracy acc =
ccBoundaryDefs::eFast) const;
```

Returns the vector of distances between the boundary points of this **ccBoundary** and the corresponding boundary points of *bnd*. The distances are computed according to the modalities specified by *method* and *acc* (see *ccBoundaryDefs*).

Parameters

bnd	The boundary from which the distance of this **ccBoundary** is measured.
method	The measurement mode used to compute boundary point distances. *method* must be one of the following: *ccBoundaryDefs::eRadial* *ccBoundaryDefs::eParallel*
acc	The matching modality used to match the boundary points of this **ccBoundary** to the boundary points of *bnd*. *acc* must be one of the following: *ccBoundaryDefs::eFast* *ccBoundaryDefs::eAccurate*

position

```
const cmStd vector <cc2Vect> &position() const;
```

Returns a list of all the positions of the boundary points in this **ccBoundary**.

angle

```
const cmStd vector <ccRadian>& angle() const;
```

Returns a list of all the direction angles of the boundary points in this **ccBoundary**.

weight

```
const cmStd vector <double>& weight() const;
```

Returns a list of all the weights of the boundary points in this **ccBoundary**.

matchQuality

```
const cmStd vector <double>& matchQuality() const;
```

Returns a list of all the matchQuality values of the boundary points in this **ccBoundary**.

ccBoundaryDefs

```
#include <ch_cvl/pmibnd.h>

class ccBoundaryDefs;
```

A name space that holds enumerations and constants used with PatInspect.

Enumerations

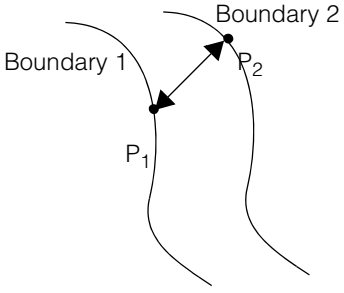
MeasurementMethod

```
enum MeasurementMethod
```

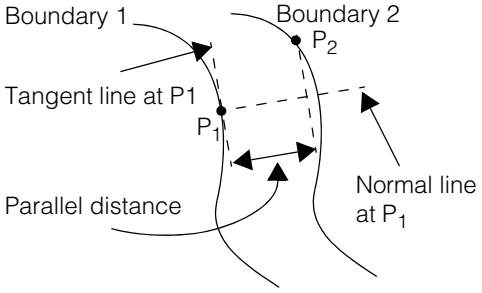
This enumeration defines the measurement methods used to compute the distance between two boundary points belonging to two distinct boundaries.

Value	Meaning
eRadial	The radial distance between boundary points is computed.
eParallel	The distance between boundary points is computed on parallel lines.

The following figure illustrates the difference between radial distance (*eRadial*) and parallel distance (*eParallel*).



Radial distance of P_2 from P_1



Parallel distance of P_2 from P_1

MeasurementAccuracy

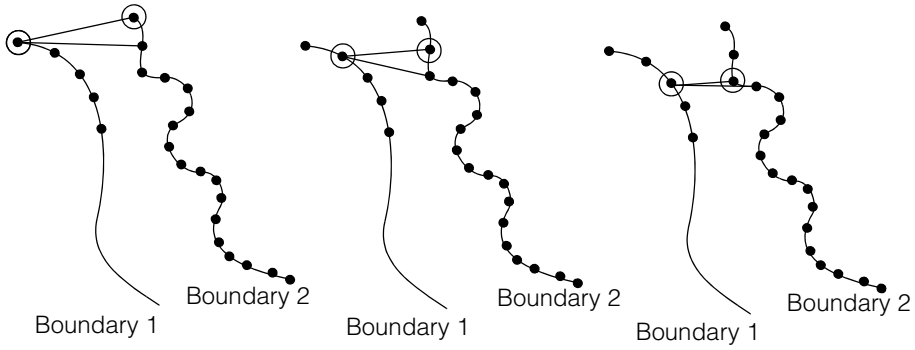
```
enum MeasurementAccuracy
```

This enumeration defines the method used to determine the matching points of two boundaries.

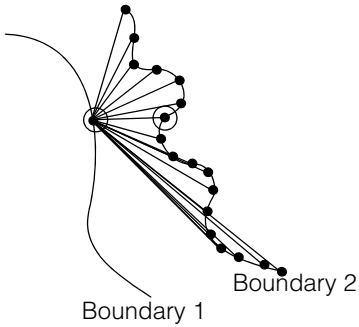
Value	Meaning
eFast	The matching is based on a local distance-comparison strategy.
eAccurate	The matching is based on an exhaustive distance-comparison strategy.

The following illustrations show the difference between the two methods:

- *eFast*
  In this mode points of Boundary 1 are matched to points of Boundary 2 on the basis of a local distance comparison strategy. The distances between a point in Boundary 1 and two points in Boundary 2 are computed and the point in Boundary 2 with the shortest distance is matched to the point in Boundary 1. The matching process is serial, starting from the first point of Boundary 1 as illustrated in the following figure (in the figure matching boundary points are circled).



- *eAccurate*
  In this mode points of Boundary 1 are matched to points of Boundary 2 on the basis of a comprehensive distance comparison strategy. A point in Boundary 1 is matched to the closest point in Boundary 2 as illustrated in the following figure (in the figure matching boundary points are circled).



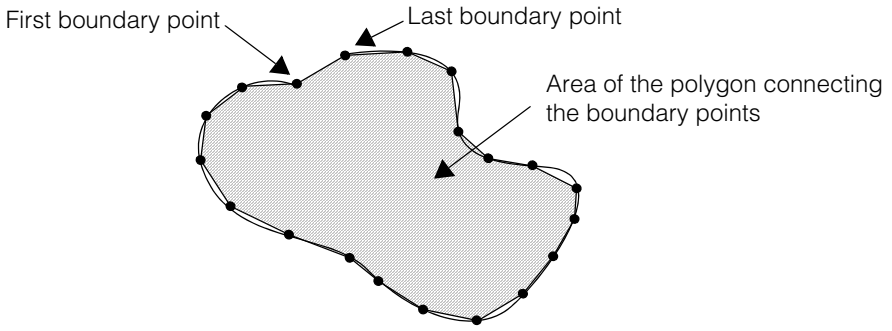
AreaMethod `enum AreaMethod`

This enumeration defines the method used to measure the area contained by an open or closed boundary.

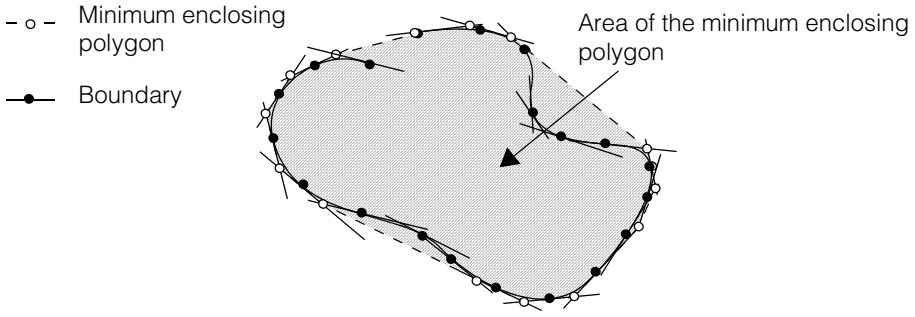
Value	Meaning
eSimple	The area of the polygon defined by the line segments between boundary points is computed.
eConvex	The area of the minimum polygon enclosing the boundary is computed.

The following illustrations show the difference between the two methods:

- *eSimple*
  The area of the polygon resulting from connecting adjacent boundary points is measured. If the boundary is open, the first and last boundary point are connected by a straight segment.



- *eConvex*
  The area of the minimum polygon enclosing the boundary is measured.



BoundaryType `enum BoundaryType`

This enumeration defines what boundary data to retain when **ccBoundary** constructors are called.

Value	Meaning
eTrained	The training boundary data associated with the matched boundary data are retained.
eMatch	The matching boundary data are retained.
eMissing	The missing boundary data that were trained but not found at run-time are retained.
eExtra	The extra boundary data that were not trained but were found at run-time are retained.

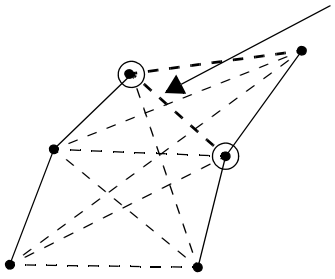
DistanceMethod `enum DistanceMethod`

This enumeration defines the method used to determine the distance between two boundaries.

Value	Meaning
eExhaustive	The distance between two boundaries is defined by the smallest distance between the boundary points.
eMinEndPnt	The distance between two boundaries is defined by the smallest distance between the end-points of the boundaries.
eMeanPosition	The distance between two boundaries is defined by the distance between the mean positions of the boundaries.

The following illustrates the difference between the three methods:

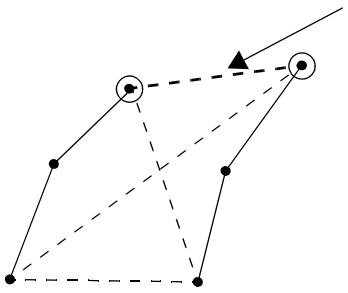
- *eExhaustive*



This is the distance between Boundary 1 and Boundary 2 according to the *eExhaustive* method. The distance between the two circled boundary points is the shortest one.

Boundary 1 Boundary 2

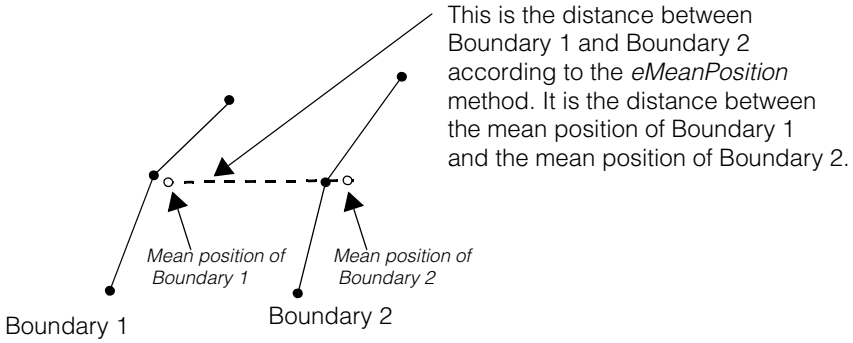
- *eMinEndPnt*



This is the distance between Boundary 1 and Boundary 2 according to the *eMinEndPnt* method. The distance between the two circled end-points is the shortest one.

Boundary 1 Boundary 2

- *eMeanPosition.*



ccBoundaryDefs

ccBoundaryInspector

```
#include <ch_cvl/bndrinsp.h>

class ccBoundaryInspector
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class encapsulates the Boundary Inspection tool. You train this tool with a **ccShape** model and then run the tool on images to determine if the image boundaries (contours) match the model boundaries. The inspection tool returns all of the matching contours and also non-matching contours in both the model and the image. The non-matching contours enable you to evaluate the match between the model and the image.

Constructors/Destructors

ccBoundaryInspector

```
ccBoundaryInspector();

ccBoundaryInspector(const ccBoundaryInspector &that);

~ccBoundaryInspector();
```

- `ccBoundaryInspector();`

  Constructs an untrained **ccBoundaryInspector** tool.

- `ccBoundaryInspector(const ccBoundaryInspector &that);`

  Copy constructor. Performs a deep copy.

- `~ccBoundaryInspector();`

  Destructor.

Operators

operator=

```
ccBoundaryInspector & operator=(
                          const ccBoundaryInspector &that);
```

Deep copy assignment operator.

operator==

```
bool operator==(const ccBoundaryInspector& that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

operator!=

```
bool operator!=(const ccBoundaryInspector& that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

Public Member Functions

train

```
void train(
  const ccShape &modelBoundary,
  const cc2XformBase &clientFromImage,
  const ccShapeTol &shapeTolerance,
  const ccBoundaryInspectorTrainParams &trainParams,
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);

void train(
  const ccShape &modelBoundary,
  const ccShapeMask &shapeMask,
  const cc2XformBase &clientFromImage,
  const ccShapeTol &shapeTolerance,
  const ccBoundaryInspectorTrainParams &trainParams,
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);
```

-
```
void train(
  const ccShape &modelBoundary,
  const cc2XformBase &clientFromImage,
```

```
const ccShapeTol &shapeTolerance,
const ccBoundaryInspectorTrainParams &trainParams,
ccDiagObject* obj = 0,
c_UInt32 diagFlags = 0);
```

Train this tool with the given parameters. The model boundary may be any CVL shape description, including a hierarchical shape. The shape description is specified in model coordinates. The relationship between the client space and image space at train time is specified by the *clientFromImage* transform. The model boundary tolerances, which are in model coordinates, are specified by the *shapeTolerance* object.

If the object is currently trained, it untrains and then trains again.

Notes

Shape descriptions or shape description portions without a defined tangent angle are ignored by this tool.

Parameters

modelBoundary A **ccShape** object that contains the model boundary.

clientFromImage The expected client transform of the images to be inspected.

shapeTolerance The allowed shape boundary tolerances. If you are doing statistical training this will be a reference to a **ccShapeTolStats** object you have created using good test images. If you are doing manual training this will be a reference to a **ccShapeTolTable** object you have created manually.

trainParams The training parameters.

obj A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccBoundaryInspectorDefs::BadParams

If the model boundary is infinite and not a primitive,
or if the model boundary is infinite and
trainParams().clipRegion() is not a valid clipping region,
or if the *shapeTolerance* object is detectably invalid for this model boundary.

ccMathError::Singular

If *clientFromImage* or its inverse is singular.

Notes

The tool becomes untrained if it throws.

- ```
  void train(
    const ccShape &modelBoundary,
    const ccShapeMask &shapeMask,
    const cc2XformBase &clientFromImage,
    const ccShapeTol &shapeTolerance,
    const ccBoundaryInspectorTrainParams &trainParams,
    ccDiagObject* obj = 0,
    c_UInt32 diagFlags = 0);
  ```

Train as described above, but include a shape mask. A shape mask is applied to the **ccShape** model and allows you to ignore (mask out) portions of the model boundary.

Parameters

modelBoundary A **ccShape** object that contains the model boundary.

shapeMask The shape mask to apply to *modelBoundary*. Note that **ccShapeTol** is a typedef as follows:

typedef
ccShapePerimData<ccShapeMaskValue> ccShapeMask

Please see the **ccShapeMaskValue** reference page for more information about shape masks.

clientFromImage The expected client transform of the images to be inspected.

shapeTolerance The allowed shape boundary tolerances. If you are doing statistical training this will be a reference to a **ccShapeTolStats** object you have created using good test images. If you are doing manual training this will be a reference to a **ccShapeTolTable** object you have created manually.

trainParams The training parameters.
  ```

obj — A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.

diagFlags — Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccBoundaryInspectorDefs::BadParams
If the model boundary is infinite and not a primitive,
or if the model boundary is infinite and
trainParams().clipRegion() is not a valid clipping region,
or if the *shapeTolerance* object or *shapeMask* are detectably invalid for this model boundary.

ccMathError::Singular
If *clientFromImage* or its inverse is singular.

Notes

The tool becomes untrained if it throws.

isTrained

```
bool isTrained() const;
```

Returns true if **train()** was called successfully on this object.

trainParams

```
const ccBoundaryInspectorTrainParams &trainParams() const;
```

Returns the **ccBoundaryInspectorTrainParams** object used in training.

Throws

ccBoundaryInspectorDefs::NotTrained
If this object is not trained.

modelBoundary

```
ccShapePtrh_const modelBoundary() const;
```

Returns the **ccShape** object used in training.

Throws
ccBoundaryInspectorDefs::NotTrained
If this object is not trained.

clientFromImage

```
cc2XformBasePtrh_const clientFromImage() const;
```

Returns the client-from-image transform used in training.

Throws
ccBoundaryInspectorDefs::NotTrained
If this object is not trained.

shapeTolerance

```
ccShapeTolPtrh_const shapeTolerance() const;
```

Returns the **ccShapeTol** object used in training.

Throws
ccBoundaryInspectorDefs::NotTrained
If this object is not trained.

shapeMask

```
ccShapeMaskPtrh_const shapeMask() const;
```

Returns the shape mask used in training. The portions of the model boundary masked by this shape mask will be ignored by the **ccBoundaryInspector** tool.

Throws
ccBoundaryInspectorDefs::NotTrained
If this object is not trained,
or if this object was not trained with a shape mask.

---

run

```
void run(
  const ccPelBuffer_const<c_UInt8> &image,
  const cc2XformBase &pose,
  ccBoundaryInspectorResult &results,
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0) const;

void run(
  const ccPelBuffer_const<c_UInt8> &image,
  const cc2XformBase &pose,
  const ccFeatureletFilter &featureletFilter,
```

```
  ccBoundaryInspectorResult &results,
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0) const;
```

- ```
  void run(
    const ccPelBuffer_const<c_UInt8> &image,
    const cc2XformBase &pose,
    ccBoundaryInspectorResult &results,
    ccDiagObject* obj = 0,
    c_UInt32 diagFlags = 0) const;
  ```

  Performs the geometric boundary inspection on the given image with the given parameters. The relationship between the client coordinates and the model coordinates of the trained boundary is specified by the *pose* transform.

  The scaling factors of *pose* should be close to 1.0 for correct operation.

  Parameters

  image — The image to be inspected.

  pose — A transform that specifies the relationship between client coordinates and the model coordinates of the trained boundary.

  results — Where to put the inspection results.

  obj — A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.

  diagFlags — Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccBoundaryInspectorDefs::BadParams
  If *image* is not bound.

  ccBoundaryInspectorDefs::NotTrained
  If this object is not trained.
  ```

ccMathError::Singular
If *pose*, **image.clientFromImageXformBase()**, or their inverses are singular.

This function will not mask any exceptions that the featurelet filter might throw.

- ```
  void run(
    const ccPelBuffer_const<c_UInt8> &image,
    const cc2XformBase &pose,
    const ccFeatureletFilter &featureletFilter,
    ccBoundaryInspectorResult &results,
    ccDiagObject* obj = 0,
    c_UInt32 diagFlags = 0) const;
  ```

  Same as above except a featurelet filter is specified. The featurelet filter is applied to the image contours after the contours are transformed into the coordinate system in which the model boundary is defined.

  See **ccFeatureletFilter** for information on featurelet filters.

  Parameters

  image — The image to be inspected.

  pose — A transform that specifies the relationship between client coordinates and the model coordinates of the trained boundary.

  featureletFilter — The featurelet filter to be applied to *image*.

  results — Where to put the inspection results.

  obj — A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.

  diagFlags — Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff
  ```

Throws

ccBoundaryInspectorDefs::BadParams
If *image* is not bound.

ccBoundaryInspectorDefs::NotTrained
If this object is not trained.

ccMathError::Singular
If *pose*, **image.clientFromImageXformBase()**, or their inverses are singular.

This function will not mask any exceptions that the featurelet filter might throw.

ccBoundaryInspector

ccBoundaryInspectorResult

```
#include <ch_cvl/bndrinsp.h>

class ccBoundaryInspectorResult
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class encapsulates the results of a boundary inspection (see **ccBoundaryInspector**). It contains both matched and unmatched results of the trained model against an image. Unmatched results include image contours, expressed as featurelet chain sets, that were not matched to the trained model. Unmatched model boundaries are expressed as perimeter ranges.

Matched image contours are also expressed as featurelet chain sets and matched model boundaries are expressed as perimeter ranges. For matched results, the class holds an array of image contours that corresponds to an array of model perimeter ranges where each image contour is a match to a corresponding model perimeter range.

Constructors/Destructors

ccBoundaryInspectorResult

```
ccBoundaryInspectorResult();
```

Constructs an empty **ccBoundaryInspectorResult** object.

Operators

operator==

```
bool operator==(
  const ccBoundaryInspectorResult& that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

operator!=

```
bool operator!=(
  const ccBoundaryInspectorResult& that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

Public Member Functions

extraImageContours

```
ccFeatureletChainSetPtrh_const extraImageContours() const;
```

Returns the extra (unmatched) image contours as a featurelet chain set, in client coordinates.

unmatchedModelBoundary

```
const cmStd vector<ccPerimRange> &unmatchedModelBoundary()
                                                    const;
```

Returns the vector of model boundary perimeter ranges that were unmatched during the inspection.

matchedImageContours

```
const cmStd vector<ccFeatureletChainSetPtrh_const> &
                                  matchedImageContours() const;
```

Returns a vector of matched image contours as a vector of featurelet chain sets. The featurelet chain set at **matchedImageContours()**[i] matches the vector of perimeter ranges at **matchedModelBoundary()**[i].

matchedModelBoundary

```
const cmStd vector<ccPerimRange> &matchedModelBoundary()
                                                    const;
```

Returns a vector of matched model boundary perimeter ranges. The vector of perimeter ranges at **matchedModelBoundary()**[i] matches the featurelet chain set at **matchedImageContours()**[i].

ccBoundaryInspectorTrainParams

```
#include <ch_cvl/bndrinsp.h>

class ccBoundaryInspectorTrainParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class encapsulates the Boundary Inspection tool training parameters. See **ccBoundaryInspector**.

There are two training parameters; *granularity*, and *clipRegion*. Granularity specifies the degree to which inspection images are smoothed and subsampled before edges are extracted. Specifying a clip region is optional. A clip region is a closed **ccPolyline** that is applied to the model before model training. Only the model boundary within the clip region is trained.

Constructors/Destructors

ccBoundaryInspectorTrainParams

```
explicit ccBoundaryInspectorTrainParams(
  double granularity = 1.0,
  const ccPolyline &clipRegion = ccPolyline(true));
```

Constructs a **ccBoundaryInspectorTrainParams** object with the given parameters.

The default granularity is 1.0, very fine granularity.

A valid clip region is a closed convex **ccPolyline** specified in model coordinates.

Parameters

granularity	The smoothing and subsampling factor applied to inspection images. Must be in the range 1.0 through 25.0. A setting of 1.0 specifies fine granularity and a setting of 25.0 specifies very coarse granularity.
clipRegion	A closed region that specifies a subset of the model for training. Use a clip region to omit parts of a model you do not wish to train.

Throws

ccBoundaryInspectorDefs::BadParams
If *granularity* < 1.0 or *granularity* > 25.0, or if the clip region is open.

Operators

operator==

```
bool operator==(
  const ccBoundaryInspectorTrainParams& that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

operator!=

```
bool operator!=(
  const ccBoundaryInspectorTrainParams& that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

granularity

```
double granularity() const;

void granularity(double granularity);
```

Granularity specifies the degree to which inspection images are smoothed and subsampled before edges are extracted. Must be in the range 1.0 through 25.0. A setting of 1.0 specifies fine granularity and a setting of 25.0 specifies very coarse granularity. The default granularity value is 1.0.

- `double granularity() const;`

  Returns the current granularity setting.

- `void granularity(double granularity);`

  Sets the granularity.

Parameters

granularity The new granularity setting.

Throws

ccBoundaryInspectorDefs::BadParams
If *granularity* < 1.0 or *granularity* > 25.0.

---

clipRegion

```
const ccPolyline& clipRegion() const;
```

```
void clipRegion(const ccPolyline& clipRegion);
```

Specifying a clipping region is optional. A valid clipping region is a closed, convex **ccPolyline** that is applied to the model before model training. Only the model boundary within the clipping region is trained. The clipping region is specified in the model coordinates.

If clipping is performed, the missing and matching model boundaries found during boundary inspection are reported only if they are within the clipping region.

No clipping is performed during training unless a valid clipping region is provided here. A valid clipping region is a closed, convex **ccPolyline** with at least 3 vertices.

The default clipping region is a closed polyline with no vertices (no clipping is performed).

- `const ccPolyline& clipRegion() const;`

  Returns the **ccPolyline** that defines the clipping region.

- `void clipRegion(const ccPolyline& clipRegion);`

  Defines a new clipping region.

  Parameters

  clipRegion The new clipping region.

  Throws

  ccBoundaryInspectorDefs::BadParams
  If the clipping region is open.

ccBoundaryInspectorTrainParams

ccBoundarySet

```
#include <ch_cvl/pmibnd.h>

class ccBoundarySet;
```

Class Properties

Copyable	Yes
Derivable	Not intended
Archiveable	Simple

This class contains the set of boundary results returned by PatInspect. The class holds data from the following four sets of boundaries:

- The boundary of trained features (the boundary features that have been trained)
- The boundary of matched features (the boundary features that are present in both run-time and template image)
- The boundary of extra features (the boundary features that are present in the run-time image but not in the template image)
- The boundary of missing features (the boundary features that are present in the template image but not in the run-time image).

Constructors/Destructors

ccBoundarySet

```
ccBoundarySet();

ccBoundarySet(ccPMInspectResult &result);

ccBoundarySet(const ccPMInspectBoundaryData &diffData,
  const cc2Xform &pose=cc2Xform::I);

ccBoundarySet(const cmStd vector <ccBoundary>& points,
  ccBoundaryDefs::BoundaryType
  type=ccBoundaryDefs::eTrained,
  const cc2Xform &pose=cc2Xform::I);

ccBoundarySet(const cmStd vector <cc2Vect>& points,
  ccBoundaryDefs::BoundaryType type,
  const cc2Xform &pose=cc2Xform::I);

ccBoundarySet(const ccPMInspectAbsenceData &points,
  const cc2Xform & pose=cc2Xform::I);
```

- `ccBoundarySet();`

  Creates a **ccBoundarySet** with empty boundaries. The boundaries of trained, matched, extra and missing features of this **ccBoundarySet** have no points.

- `ccBoundarySet(ccPMInspectResult &result);`

  Creates the boundaries of trained, matched, extra and missing features of this **ccBoundarySet** from the corresponding boundaries returned by *result*. The four boundaries are expressed in client coordinates. The transformation that maps the boundaries to client coordinates is included in *results*.

  Parameters

  result — The **ccPMInspectResult** object to which this **ccBoundarySet** is initialized.

- `ccBoundarySet(const ccPMInspectBoundaryData &diffData, const cc2Xform &pose=cc2Xform::I);`

  Creates the boundaries of trained, matched, extra and missing features of this **ccBoundarySet** from the corresponding boundaries returned by *diffData*. The four boundaries are expressed in client coordinates. The transformation that maps the boundaries to client coordinates is provided by *pose*.

  Parameters

  diffData — The **ccPMInspectBoundaryData** object to which this **ccBoundarySet** is initialized.

pose The **cc2Xform** that maps the boundary features to client coordinates.

- ```
  ccBoundarySet(const cmStd vector <ccBoundary>& points,
    ccBoundaryDefs::BoundaryType
    type=ccBoundaryDefs::eTrained,
    const cc2Xform &pose=cc2Xform::I);
  ```

  Creates the boundary specified by *type* of this **ccBoundarySet** from the vector of **ccBoundary** *points*. The three remaining boundaries are all initialized to a boundary with no points. The transformation that maps the boundaries to client coordinates is provided by *pose*.

  Parameters

  points The vector of **ccBoundary** points used to initialize this **ccBoundarySet**.

  type The modifier that specifies which boundary of this **ccBoundarySet** is set to *points*. *type* must be one of the following:
  ccBoundaryDefs::eTrainedt
  ccBoundaryDefs::eMatch
  ccBoundaryDefs::eMissing
  ccBoundaryDefs::eExtra

  pose The **cc2Xform** that maps the boundary features to client coordinates.

- ```
  ccBoundarySet(const cmStd vector <cc2Vect>& points,
    ccBoundaryDefs::BoundaryType type,
    const cc2Xform &pose=cc2Xform::I);
  ```

  Sets the boundary specified by *type* of this **ccBoundarySet** to the vector of points *points*. The three remaining boundaries are all initialized to a boundary with no points. The transformation that maps the boundaries to client coordinates is provided by *pose*.

  Parameters

  points The vector of points used to initialize this **ccBoundarySet**.

  type The modifier that specifies which boundary of this **ccBoundarySet** is set to *points*.

  pose The **cc2Xform** that maps the boundary features to client coordinates.

- `ccBoundarySet(const ccPMInspectAbsenceData &points, const cc2Xform & pose=cc2Xform::I);`

  Constructs a **ccBoundarySet** from the supplied **ccPMInspectAbsenceData** object. The points are transformed by the supplied pose.

  Parameters

  points — The **ccPMInspectAbsenceData** used to initialize this **ccBoundarySet**.

  pose — The **cc2Xform** that maps the boundary features to client coordinates.

Public Member Functions

draw

```
void draw(ccUITablet &tablet,
  bool connected=true,
  const ccColor &trainedColor =ccColor::blueColor(),
  const ccColor &matchColor =ccColor::greenColor(),
  const ccColor &missingColor =ccColor::yellowColor(),
  const ccColor &extraColor=ccColor::redColor(),
  const ccUITablet::Layers &layer=
  ccUITablet::eImageLayer) const;
```

Draws all the boundaries of this **ccBoundarySet** in the supplied **ccUITablet** *tablet*.

Parameters

tablet — The tablet used to draw this **ccBoundarySet**.

connected — If true the function draws the polygon formed by connecting the adjacent boundary points of this **ccBoundarySet**.

trainedColor — The color of the trained boundary features of this **ccBoundarySet**.

matchColor — The color of the matching boundary features of this **ccBoundarySet**.

missingColor — The color of the missing boundary features of this **ccBoundarySet**.

extraColor — The color of the extra boundary features of this **ccBoundarySet.**

layer — The layer into which this **ccBoundarySet** is drawn.

boundaryVect

```
cmStd vector <ccBoundary> & boundaryVect(
  ccBoundaryDefs::BoundaryType
  type=ccBoundaryDefs::eTrained);

const cmStd vector <ccBoundary> & boundaryVect(
  ccBoundaryDefs::BoundaryType
  type=ccBoundaryDefs::eTrained) const;
```

- ```
  cmStd vector <ccBoundary> & boundaryVect(
    ccBoundaryDefs::BoundaryType
    type=ccBoundaryDefs::eTrained);
  ```

  Returns the boundary of this **ccBoundarySet** specified by *type*.

  Parameters

type	The modifier that specifies which boundary of this **ccBoundarySet** is returned by the function.

- ```
  const cmStd vector <ccBoundary> & boundaryVect(
    ccBoundaryDefs::BoundaryType
    type=ccBoundaryDefs::eTrained) const;
  ```

  Returns the boundary of this **ccBoundarySet** specified by *type*.

  Parameters

type	The modifier that specifies which boundary of this **ccBoundarySet** is returned by the function.

ccBoundarySet

ccBoundaryTol

```
#include <ch_cvl/shapetol.h>

class ccBoundaryTol
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

ccBoundaryTol is a container class for representing the acceptable (tolerable) positional and angular difference between a shape boundary point and a 2D point with a tangent angle. The Boundary Inspection tool requires that you provide acceptable variations of the boundaries to be inspected. These acceptable variations are called *shape tolerances*. **ccBoundaryTol** is used as a data type to instantiate the templated classes **ccShapePerimData** and **ccShapePerimDataTable** for creating the **ccShapeTol** and **ccShapeTolTable** classes. Typedefs creating the **ccShapeTol** and **ccShapeTolTable** classes are located on the **ccShapePerimData** and **ccShapePerimDataTable** reference pages.

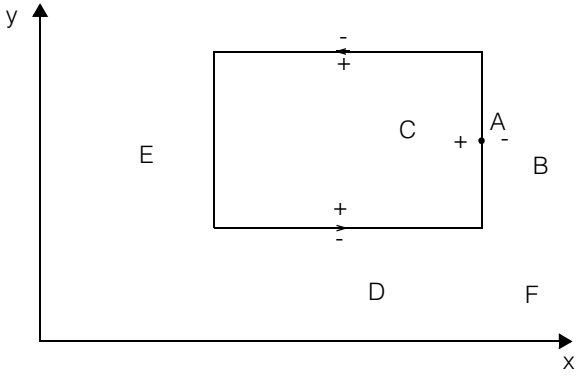
The absolute value of the positional difference between a boundary point and a 2D point is the Euclidean distance between them. If the boundary point has a defined tangent angle, then the positional difference has a sign which is positive if the 2D point is on the positive side of the boundary point and negative if the 2D point is on the negative side. In standard CVL notation, the positive side of a boundary point is the +90° direction from the tangent direction of the boundary point. Similarly, the negative side is the -90° direction from the tangent. All the points that are on the tangent line of the boundary point are assumed to have positive distance signs. Please see comments for **ccShape** for a description of tangent direction.

In a right-handed coordinate system, the +90° and -90° directions correspond to counterclockwise and clockwise rotations from the tangent vector, respectively.

The above definition of positive and negative side implies that the inside regions of right-handed closed shape descriptions (all primitive shape regions such as circles and rectangles) are the positive sides, and outside regions are the negative sides.

As an example, consider the rectangle and the boundary point A shown in the right-handed coordinate system below. The distances from boundary point A to points B and F are negative because the points B and F are in the +90° direction side from the

tangent direction of boundary point A. The points C, D, and E are a positive distance away from the boundary point A. Note that being inside or outside of a boundary region does not determine the sign of the distance to the point by itself.



Positive and negative sides of a shape should not be confused with the shape polarity defined by **ccShapeModelProps**. In particular, the positive and negative sides of a shape depend only on the shape tangent direction and cannot be changed by reversing the shape model polarity flag.

The angular difference between the angles of a boundary point and a 2D point is calculated depending on the *ignorePolarity* flag. If the *ignorePolarity* flag is false, the angular difference is calculated by taking the absolute value of the difference of the angles in the range 0 through pi. If the *ignorePolarity* flag is true, the angles are first mapped to the range 0 through pi, and then the difference is taken in range 0 through pi/2.

Constructors/Destructors

ccBoundaryTol

```
explicit ccBoundaryTol(
  const ccRange & distanceTol = ccRange::FullRange(),
  const ccRadian &angleTol = ccRadian(ckPI),
  bool ignorePolarity = false);
```

Constructs a **ccBoundaryTol** object with the given parameters.

Parameters

distanceTol	The maximum tolerable distance between a model boundary point and a corresponding image point.
angleTol	The maximum tolerable angular difference between a model boundary featurelet and a corresponding image featurelet.

ignorePolarity If true, polarity is ignored when calculating the featurelet angular difference described above. If false, featurelet polarity is used in the calculation.

Throws

ccShapePerimDataDefs::BadParams
If *angleTol* < **ccRadian(**0.0**)**.

Operators

operator==

```
bool operator== (const ccBoundaryTol &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

operator!=

```
bool operator!= (const ccBoundaryTol &that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

distanceTol

```
ccRange distanceTol() const;
```

```
void distanceTol(const ccRange &distanceTol);
```

The maximum tolerable distance between a model boundary point and a corresponding image point.

The default value is **ccRange::FullRange()**.

- `ccRange distanceTol() const;`

  Returns the current distance tolerance.

- `void distanceTol(const ccRange &distanceTol);`

  Sets a new distance tolerance.

Parameters
distanceTol The new distance tolerance.

angleTol

```
ccRadian angleTol() const;
void angleTol(const ccRadian &angleTol);
```

The maximum tolerable angular difference between a model boundary featurelet and a corresponding image featurelet.

The default angle tolerance value is **ccRadian(**ckPI**)**.

- `ccRadian angleTol() const;`

  Returns the current angle tolerance.

- `void angleTol(const ccRadian &angleTol);`

  Sets a new angle tolerance.

  Parameters
  angleTol The new angle tolerance.

  Throws
  ccShapePerimDataDefs::BadParams
  If *angleTol* < **ccRadian(**0.0**)**.

  Note: The state of this object remains unchanged if it throws.

ignorePolarity

```
bool ignorePolarity() const;
void ignorePolarity(bool ignorePolarity);
```

If true, polarity is ignored when calculating the featurelet angular difference described above for angle tolerance. If false, featurelet polarity is used in the calculation.

The default ignorePolarity flag value is false.

- `bool ignorePolarity() const;`

  Returns the state of the ignore polarity flag, true or false.

- `void ignorePolarity(bool ignorePolarity);`

  Sets the ignore polarity flag, true of false.

Parameters

ignorePolarity The new ignore polarity flag.

ccBoundaryTol

ccBoundaryTrackerDefs

```
#include <ch_cvl/bndtrckr.h>

class ccBoundaryTrackerDefs;
```

A name space that holds enumerations and constants used with the Symbol tool classes.

Enumerations

ModeFlags

```
enum ModeFlags;
```

This enumeration defines the Boundary Tracker tool modes.

Value	Meaning
eWhiteOnBlack	Specifies light object on dark background (default is dark object on light background)
eGradientThreshold	Specifies gradient tracking (default is threshold tracking)
eStopAtImageEdge	Stop tracking at image edge (default is to return to start point and tracking in the other direction)
kDefaulttFlags	Default modes (dark object on light background; threshold tracking; re-start tracking if image edge encountered)

ccBoundaryTrackerDefs

ccBoundaryTrackerPoint

```
#include <ch_cvl/bndtrckr.h>

class ccBoundaryTrackerPoint;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

A class that describes a single boundary point used by the Boundary Tracker tool. A boundary point contains a subpixel location and a direction.

Constructors/Destructors

ccBoundaryTrackerPoint

```
ccBoundaryTrackerPoint();

ccBoundaryTrackerPoint(const cc2Vect& c,
  const ccRadian& a);
```

- `ccBoundaryTrackerPoint();`

  Constructs a **ccBoundaryTrackerPoint** with a location of (0, 0) and an angle of 0.

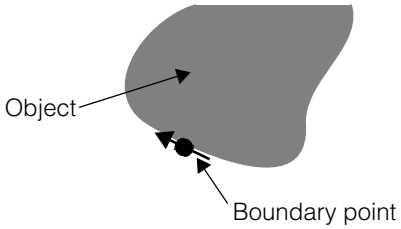
- ```
  ccBoundaryTrackerPoint(const cc2Vect& c,
    const ccRadian& a);
  ```

  Constructs a **ccBoundaryTrackerPoint** using the supplied location and angle.

  Parameters

c	The location of the **ccBoundaryTrackerPoint**.
  ```

a The angle of the **ccBoundaryTrackerPoint**. A **ccBoundaryTrackerPoint**'s angle is tangent to the object boundary and in the tracking direction, as shown in this figure:



A **ccBoundaryTrackerPoint**'s angle is in the image's client coordinate system.

Operators

operator== `bool operator==(const ccBoundaryTrackerPoint& rhs);`

Two **ccBoundaryTrackerPoint**s are equal if their locations and angles are the same.

ccBoundaryTrackerResult

```
#include <ch_cvl/bndtrckr.h>

class ccBoundaryTrackerResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

A class that contains information about a single object boundary measured by the Boundary Tracker tool. You supply an object of type **ccBoundaryTrackerResult** to the function **cfBoundaryTracker()**.

Constructors/Destructors

ccBoundaryTrackerResult

```
ccBoundaryTrackerResult();
```

Constructs a **ccBoundaryTrackerResult** with no result information.

Enumerations

StatusFlags

```
enum StatusFlags;
```

This enumeration defines the status flags returned by **statusFlags()**.

Value	Meaning
eSuccess	Valid boundary was found and tracked.
eOpenBoundary	Boundary track did not return to start point.

Value	Meaning
eMaxPoints	Boundary tracking stopped because the maximum number of points were tracked. If the tool is tracking an open boundary that extends to the edge of the image, this flag may not be set until up to twice the maximum specified number of points have been tracked.
eWeakGradient	Boundary tracking stopped because gradient fell below threshold.

Public Member Functions

area

```
double area() const;
```

Returns the area of the object enclosed by this boundary. The area is the number of pixels within the boundary, transformed into client coordinate system units.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

boundaryPoints

```
const cmStd vector<ccBoundaryTrackerPoint>&
  boundaryPoints() const;
```

Returns a vector of **ccBoundaryTrackerPoint**s describing the boundary. The angle information in the returned **ccBoundaryTrackerResult** is valid only if **isAngleDataValid()** returns true.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

centerOfMass

```
const cc2Vect& centerOfMass() const;
```

Returns the center of mass of the object enclosed by this boundary in client coordinates.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

physExtent `const ccRect& physExtent() const;`

Returns the smallest rectangle that both is aligned in the image coordinate system and completely encloses this boundary. The rectangle is returned in client coordinates.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

firstBoundaryPointIndex
`c_Int32 firstBoundaryPointIndex() const;`

Returns the index into the vector of boundary points returned by **boundaryPoints()** of the first boundary point found by the Boundary Tracker tool. If you used track-only mode, this point is the same point you specified as the starting point.

If this function returns 0, it means that the second side of the boundary was not tracked.

Notes

Due to the subpixel interpolation, the coordinates of the returned point may differ slightly from the coordinates the starting point you supplied in track-only mode.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

trackTime `double trackTime() const;`

Returns the time, in seconds, it took to find and track this boundary.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

dominantAngle `const ccRadian& dominantAngle() const;`

Returns the dominant (peak) angle from the angle histogram.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked or no angle data was computed for this boundary.

Notes

This function only returns valid data if you have computed angle data for this boundary.

dominantFoldedAngle

```
const ccRadian& dominantFoldedAngle() const;
```

Returns the dominant (peak) angle from the folded angle histogram.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked or no angle data was computed for this boundary.

Notes

This function only returns valid data if you have computed angle data for this boundary.

xProjection

```
const cmStd vector<c_Int32>& xProjection() const;
```

Returns the x-projection of the portion of the input image enclosed by the smallest rectangle that both is aligned in the image coordinate system and completely encloses this boundary.

The returned 1-dimensional image is the same size as the x-dimension of the enclosing rectangle. Each pixel in the returned image is set to the sum of all of the object pixels in the corresponding column of the input rectangle.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked.

Notes

This function only returns valid data if you have computed angle data for this boundary.

yProjection

```
const cmStd vector<c_Int32>& yProjection() const;
```

Returns the y-projection of the portion of the input image enclosed by the smallest rectangle that both is aligned in the image coordinate system and completely encloses this boundary.

The returned 1-dimensional image is the same size as the y-dimension of the enclosing rectangle. Each pixel in the returned image is set to the sum of all of the object pixels in the corresponding row of the input rectangle.

Throws

ccBoundaryTrackerDefs::UninitializedData
No valid boundary has been tracked or no angle data was computed for this boundary.

Notes

This function only returns valid data if you have computed angle data for this boundary.

isAngleDataValid

```
bool isAngleDataValid() const;
```

Returns true if angle information has been computed for this boundary, false otherwise. Whether or not angles are computed is controlled by the **ccBoundaryTrackerRunParams** used to track this boundary.

statusFlags

```
c_UInt32 statusFlags() const;
```

Returns the result status flags for this boundary. The returned value is computed by ORing together one or more of the following values:

ccBoundaryTrackerResult::eSuccess
ccBoundaryTrackerResult::eOpenBoundary
ccBoundaryTrackerResult::eMaxPoints
ccBoundaryTrackerResult::eWeakGradient
ccBoundaryTrackerResult::eInternalError

ccBoundaryTrackerResult

ccBoundaryTrackerRunParams

```
#include <ch_cvl/bndtrckr.h>

class ccBoundaryTrackerRunParams: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that contains the run-time parameters for the Boundary Tracker tool. You must specify whether this **ccBoundaryTrackerRunParams** is for track-only or search-and-track mode use at construction-time. You supply an object of type **ccBoundaryTrackerRunParams** to the function **cfBoundaryTracker()**.

Constructors/Destructors

ccBoundaryTrackerRunParams

```
ccBoundaryTrackerRunParams();

ccBoundaryTrackerRunParams(
  const ccBoundaryTrackerPoint& firstBoundaryPoint,
  c_Int32 threshold,
  c_UInt32 flags = ccBoundaryTrackerDefs::kDefaultFlags,
  bool decreasingAngleSideFirst = true,
  c_Int32 maxBoundaryPoints = 0,
  const ccRange& areaRange = ccRange(0, HUGE_VAL),
  c_Int32 preAnglePoints = 1, c_Int32 postAnglePoints = 1);

ccBoundaryTrackerRunParams(
  const cmStd vector<ccLineSeg>& searchVectors,
  c_Int32 threshold,
  c_UInt32 flags = ccBoundaryTrackerDefs::kDefaultFlags,
  bool decreasingAngleSideFirst = true,
  c_Int32 maxBoundaryPoints = 0,
  const ccRange& areaRange = ccRange(0, HUGE_VAL),
  c_Int32 preAnglePoints = 1, c_Int32 postAnglePoints = 1);
```

- `ccBoundaryTrackerRunParams();`

  Constructs a **ccBoundaryTrackerRunParams** that cannot be used to track object boundaries.

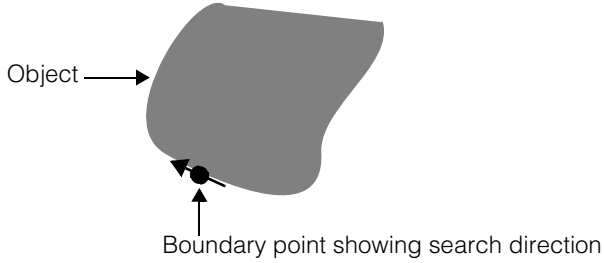
- ```
  ccBoundaryTrackerRunParams(
    const ccBoundaryTrackerPoint& firstBoundaryPoint,
    c_Int32 threshold,
    c_UInt32 flags = ccBoundaryTrackerDefs::kDefaultFlags,
    bool decreasingAngleSideFirst = true,
    c_Int32 maxBoundaryPoints = 0,
    const ccRange& areaRange = ccRange(0, HUGE_VAL),
    c_Int32 preAnglePoints = 1, c_Int32 postAnglePoints = 1);
  ```

  Constructs a **ccBoundaryTrackerRunParams** for track-only mode. For track-only mode, you must specify a starting point on the object boundary; the tool does not search for the boundary.

  Parameters

  firstBoundaryPoint

  The starting boundary point. *firstBoundaryPoint* is given in client coordinates and must lie within a boundary pixel. The angle component of *firstBoundaryPoint* must specify an angle that is tangent to the boundary in the direction of the search, as shown in the following figure:

  threshold

  The threshold value to use for object tracking. If *flags* specifies gradient tracking mode, then *threshold* is interpreted as the gradient threshold (a pixel value difference). Otherwise, *threshold* is treated as a hard binary threshold.

  If *threshold* is a hard binary threshold and *flags* specifies a light object on a dark background, then *threshold* is the minimum pixel value for an object pixel. If *threshold* is a hard binary threshold and *flags* specifies a dark object on a light background, then *threshold* is the maximum pixel value for an object pixel.

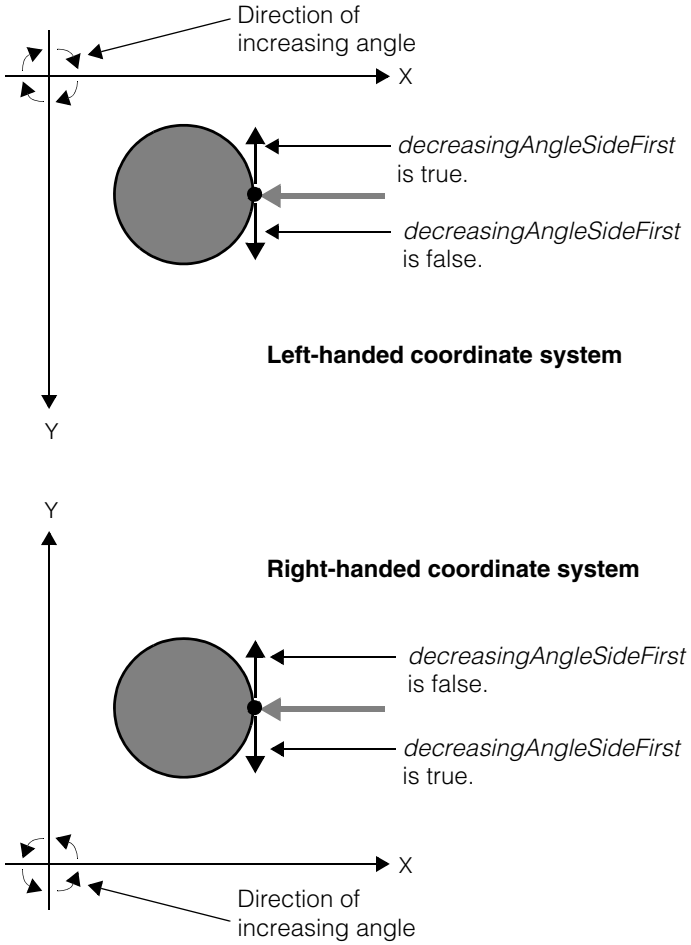
  flags

  The mode flags for this **ccBoundaryTrackerRunParams**. *flags* must be either *ccBoundaryTrackerDefs::kDefaultFlags* or a value formed by ORing together one or more of the following values:
  ```

ccBoundaryTrackerDefs::eWhiteOnBlack
ccBoundaryTrackerDefs::eGradientThreshold
ccBoundaryTrackerDefs::eStopAtImageEdge

decreasingAngleSideFirst

Set *decreasingAngleSideFirst* to true to track the boundary in the direction of decreasing client coordinate system angle. Set *decreasingAngleSideFirst* to false to track the boundary in the direction of increasing client coordinate system angle. The tracking direction is illustrated in the following figure:



maxBoundaryPoints

Specifies the maximum number of boundary points to track. The Boundary Tracker tool stops tracking after it tracks the specified number of points.

If the tool is tracking an open boundary that extends to the edge of the image, the tool may stop after it tracks up to twice as many points as you specify for *maxBoundaryPoints*.

areaRange Specifies the minimum and maximum object area size. Object sizes are positive while hole sizes are negative. To prevent the tool from tracking holes, specify a minimum object size greater than or equal to zero. To prevent the tracking of open boundaries, specify an *areaRange* that does not include 0.

For more information, see the *areaRange* definition for the **ccBoundaryTrackerRunParams** object for search-and-track mode.

preAnglePoints Specifies the number of boundary points preceding each boundary point that are used to compute that point's angle. The more points you specify, the more accurate the angle computation.

postAnglePoints Specifies the number of boundary points following each boundary point that are used to compute that point's angle. The more points you specify, the more accurate the angle computation.

Specify 0 for both *preAnglePoints* and *postAnglePoints* to not compute any angle information.

Throws

ccBoundaryTrackerDefs::ThresholdOutOfRange
threshold is greater than 255 or less than 0.

ccBoundaryTrackerDefs::BadFlags
An invalid value was specified for *flags*.

ccRangeDefs::BadParams
The value returned by **areaRange.start()** is greater than that returned by **areaRange.end()**.

ccBoundaryTrackerDefs::BadPrePostAnglePoints
preAnglePoints or *postAnglePoints* is less than 0.

ccBoundaryTrackerDefs::MaxBoundaryPointsOutOfRange
maxBoundaryPoints is less than 0.

- ```
  ccBoundaryTrackerRunParams(
    const cmStd vector<ccLineSeg>& searchVectors,
    c_Int32 threshold,
    c_UInt32 flags = ccBoundaryTrackerDefs::kDefaultFlags,
    c_Int32 maxBoundaryPoints = 0,
    const ccRange& areaRange = ccRange(0, HUGE_VAL),
    c_Int32 preAnglePoints = 1, c_Int32 postAnglePoints = 1);
  ```

  Constructs a **ccBoundaryTrackerRunParams** object for search-and-track mode. In search-and-track mode, you provide the tool with search parameters using this object. The tool then returns the first boundary that meets all specified conditions.

  The tool tracks any boundary that it finds, and then evaluates whether the boundary meets the specified conditions. If the boundary does not meet these conditions, the tool does not return data about the boundary and continues its search for a boundary that does.

  Parameters

  searchVectors — A vector of search lines. Each search line should start inside an object and extend across the object boundary. Each search line is searched for transitions of the correct polarity that exceed the specified threshold value. Each transition is used as a start point for a boundary track.

  threshold — The threshold value to use for object tracking. If *flags* specifies gradient tracking mode, then *threshold* is interpreted as the gradient threshold (a pixel value difference). Otherwise, *threshold* is treated as a hard binary threshold.

  If *threshold* is a hard binary threshold and *flags* specifies a light object on a dark background, then *threshold* is the minimum pixel value for an object pixel. If *threshold* is a hard binary threshold and *flags* specifies a dark object on a light background, then *threshold* is the maximum pixel value for an object pixel.

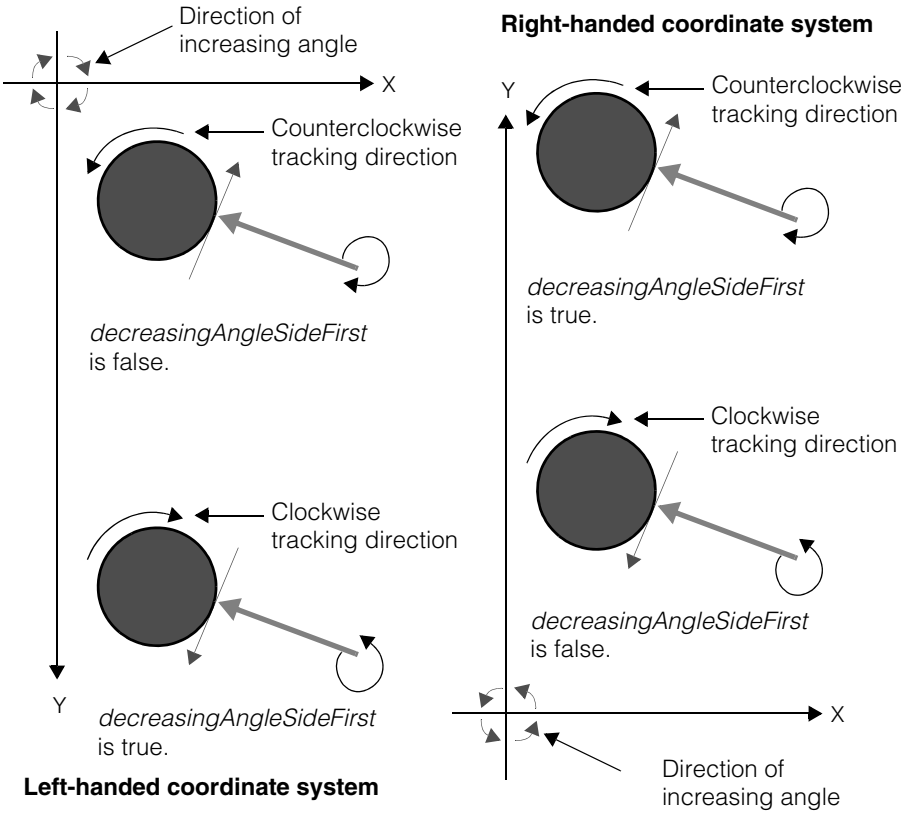
  flags — The mode flags for this **ccBoundaryTrackerRunParams**. *flags* must be either *ccBoundaryTrackerDefs::kDefaultFlags* or a value formed by ORing together one or more of the following values:

  ccBoundaryTrackerDefs::eWhiteOnBlack
  ccBoundaryTrackerDefs::eGradientThreshold
  c*cBoundaryTrackerDefs::eStopAtImageEdge*

  decreasingAngleSideFirst
  Set *decreasingAngleSideFirst* to true to track the boundary in the direction of decreasing client coordinate system angle. Set *decreasingAngleSideFirst* to false to track the boundary in the direction of increasing client coordinate system angle. The
  ```

decreasingAngleSideFirst parameter refers to the direction of the angle created if the search vector were to be rotated in an increasing or decreasing angle direction in the client coordinate system.

The tracking direction is illustrated in the following figure:



maxBoundaryPoints

Specifies the maximum number of boundary points to track. The Boundary Tracker tool will stop tracking after it tracks the specified number of points.

If the tool is tracking an open boundary that extends to the edge of the image, the tool may stop after it tracks up to twice as many points as you specify for *maxBoundaryPoints*.

areaRange Specifies the minimum and maximum object area size. Object areas are positive; hole areas are negative; and the areas of open boundaries equal 0.

Use the following table to determine the ranges that you can specify to track objects, holes, or open boundaries:

Region 1: Holes	Region 2: Open Boundaries	Region 3: Objects
areaRange < 0 (Minimum and maximum < 0.)	*areaRange = 0* (Minimum and maximum = 0.)	*areaRange > 0* (Minimum and maximum = 0.)
Track the boundaries of holes only.	Track open boundaries only.	Track the boundaries of objects only.

To track a combination objects, create ranges with overlapping regions. The following table provides examples of different ranges:

Range	Meaning
range [1 to 10000]	Track closed object boundaries only. (Do not track nonhole and open boundaries.)
range [0 to 0]	Track open boundaries only.
range [0 to 10000]	Track open and closed boundaries, but not holes.
range [-100 to 200]	Track holes whose area is smaller than 100, open boundaries, and object boundaries whose area is smaller than 200.

To prevent the tool from tracking holes, specify a minimum object size greater than or equal to zero. To prevent the tracking of open boundaries, specify that *areaRange* not include 0.

preAnglePoints Specifies the number of boundary points preceding each boundary point that are used to compute that point's angle.

postAnglePoints Specifies the number of boundary points following each boundary point that are used to compute that point's angle.

Specify 0 for both *preAnglePoints* and *postAnglePoints* to not compute any angle information.

Throws

ccBoundaryTrackerDefs::ThresholdOutOfRange
threshold is greater than 255 or less than 0.

ccBoundaryTrackerDefs::BadFlags
An invalid value was specified for *flags*.

ccRangeDefs::BadParams
The value returned by *areaRange.start()* is greater than that returned by *areaRange.end()*

ccBoundaryTrackerDefs::BadPrePostAnglePoints
preAnglePoints or *postAnglePoints* is less than 0.

ccBoundaryTrackerDefs::MaxBoundaryPointsOutOfRange
maxBoundaryPoints is less than 0.

ccBoundaryTrackerDefs::NoSearchVectors
searchVectors contains no elements.

Public Member Functions

searchVectors

```
const cmStd vector<ccLineSeg>& searchVectors() const;

void searchVectors(
  const cmStd vector<ccLineSeg>& searchVectors);
```

- `const cmStd vector<ccLineSeg>& searchVectors() const;`

  Returns the vector of search lines for this **ccBoundaryTrackerRunParams**. This **ccBoundaryTrackerRunParams** must have been constructed using the search-and-track mode constructor.

  Throws

  ccBoundaryTrackerDefs::UninitializedData
  This object was not constructed using the search-and-track mode constructor.

- ```
  void searchVectors(
    const cmStd vector<ccLineSeg>& searchVectors);
  ```

  Sets the vector of search lines for this **ccBoundaryTrackerRunParams**. This **ccBoundaryTrackerRunParams** must have been constructed using the search-and-track mode constructor.
  ```

Parameters

searchVectors A vector of search lines. A search line can start inside or outside of an object. Along each search line, the tool looks for transitions of the correct polarity that exceed the specified threshold value. Each transition is used as a start point for a boundary track.

Throws

ccBoundaryTrackerDefs::UninitializedData
This object was not constructed using the search-and-track mode constructor.

decreasingAngleSideFirst

```
bool decreasingAngleSideFirst() const;

void decreasingAngleSideFirst(
  bool decreasingAngleSideFirst);
```

- `bool decreasingAngleSideFirst() const;`

  Returns true if the tracking direction is in the decreasing client coordinate system angle direction, false if the tracking direction is in the increasing client coordinate system angle direction.

- `void decreasingAngleSideFirst(`
  `bool decreasingAngleSideFirst);`

  Sets the boundary tracking direction.

  Parameters

  decreasingAngleSideFirst
  Specify true to track in the decreasing client coordinate system angle direction, false to track in the increasing client coordinate system direction.

threshold

```
c_Int32 threshold() const;

void threshold(c_Int32 threshold);
```

- `c_Int32 threshold() const;`

  Returns the threshold value for object pixels.

- `void threshold(c_Int32 threshold);`

  Sets the threshold value for object pixels.

  Parameters

  threshold The threshold value to use for object tracking. If *flags* specifies gradient tracking mode, then *threshold* is interpreted as the gradient threshold (a pixel value difference). Otherwise, *threshold* is treated as a hard binary threshold.

  Throws

  ccBoundaryTrackerDefs::ThresholdOutOfRange
  threshold is greater than 255 or less than 0.

---

flags

```
c_UInt32 flags() const;

void flags(c_UInt32 flags);
```

- `c_UInt32 flags() const;`

  Returns the mode flags for this **ccBoundaryTrackerRunParams**. The returned value is either *ccBoundaryTrackerDefs::kDefaultFlags* or a value formed by ORing together one or more of the following values:

  ccBoundaryTrackerDefs::eWhiteOnBlack
  ccBoundaryTrackerDefs::eGradientThreshold
  c*cBoundaryTrackerDefs::eStopAtImageEdge*

- `void flags(c_UInt32 flags);`

  Sets the mode flags for this **ccBoundaryTrackerRunParams**.

  Parameters

  flags The flags to set. *flags must be* either *ccBoundaryTrackerDefs::kDefaultFlags* or a value formed by ORing together one or more of the following values:

  ccBoundaryTrackerDefs::eWhiteOnBlack
  ccBoundaryTrackerDefs::eGradientThreshold
  c*cBoundaryTrackerDefs::eStopAtImageEdge*

  Throws

  ccBoundaryTrackerDefs::BadFlags
  flags is not a legal value, as described above.

areaRange

```
const ccRange& areaRange() const;
void areaRange(const ccRange& areaRange);
```

- `const ccRange& areaRange() const;`

  Returns the minimum and maximum object area, in client units, for this **ccBoundaryTrackerRunParams**. In track-only mode, the tool will only return a boundary track if it describes an object within the specified size range. In search-and-track mode, the tool tracks boundaries until it finds an object within the specified limits.

- `void areaRange(const ccRange& areaRange);`

  Sets the minimum and maximum object area, in client units, for this **ccBoundaryTrackerRunParams**. In track-only mode, the tool returns a boundary track only if it describes an object within the specified size range. In search-and-track mode, the tool tracks boundaries until it finds an object within the specified limits.

  The minimum area is equal to the start of the range and the maximum area is equal to the end of the range.

  Object sizes are positive while hole sizes are negative. To prevent the tool from tracking holes, specify a minimum object size greater than or equal to zero. To prevent the tracking of open boundaries, specify a maximum object size of 0.

  Parameters

  areaRange The minimum and maximum object size, in client units.

preAnglePoints

```
c_Int32 preAnglePoints() const;
void preAnglePoints(c_Int32 preAnglePoints);
```

- `c_Int32 preAnglePoints() const;`

  Returns the number of boundary points preceding each boundary point that are used to compute that point's angle.

- `void preAnglePoints(c_Int32 preAnglePoints);`

  Sets the number of boundary points preceding each boundary point that are used to compute that point's angle.

  Parameters

  preAnglePoints The number of points.

Throws

ccBoundaryTrackerDefs::BadPrePostAnglePoints
preAnglePoints is less than 0.

Notes

Specify 0 for both the number of preceding and the number of following points to suppress the calculation of angle information for boundary points.

---

postAnglePoints

```
c_Int32 postAnglePoints() const;
void postAnglePoints(c_Int32 postAnglePoints);
```

- `c_Int32 postAnglePoints() const;`

  Returns the number of boundary points following each boundary point that are used to compute that point's angle.

- `void postAnglePoints(c_Int32 postAnglePoints);`

  Sets the number of boundary points following each boundary point that are used to compute that point's angle.

  Parameters

  postAnglePoints The number of points.

  Throws

  ccBoundaryTrackerDefs::BadPrePostAnglePoints
  preAnglePoints is less than 0.

  Notes

  Specify 0 for both the number of preceding and the number of following points to suppress the calculation of angle information for boundary points.

maxBoundaryPoints

```
c_Int32 maxBoundaryPoints() const;
```

Returns the maximum number of boundary points that will be tracked. This parameter can only be set at construction.

If the tool is tracking an open boundary that extends to the edge of the image, the tool may stop after it tracks up to twice as many points as you specify for *maxBoundaryPoints*.

searchAndTrack `bool searchAndTrack() const;`

Returns true if this **ccBoundaryTrackerRunParams** was constructed using the search-and-track mode constructor. Returns false if this **ccBoundaryTrackerRunParams** was constructed using the track-only mode constructor.

firstBoundaryPoint

```
const ccBoundaryTrackerPoint& firstBoundaryPoint() const;

void firstBoundaryPoint(
  const ccBoundaryTrackerPoint& firstBoundaryPoint);
```

- `const ccBoundaryTrackerPoint& firstBoundaryPoint() const;`

  Returns the starting boundary tracking point for this **ccBoundaryTrackerRunParams**. This **ccBoundaryTrackerRunParams** must have been constructed using the track-only mode constructor.

  Throws

  ccBoundaryTrackerDefs::UnintializedData,
  This object was constructed with search-and track interface and no successful tracking has been done

  Notes

  If this **ccBoundaryTrackerRunParams** was constructed using the track-only mode constructor, this function will return the same starting point that was supplied to the constructor even if no valid boundary track was found.

- ```
  void firstBoundaryPoint(
    const ccBoundaryTrackerPoint& firstBoundaryPoint);
  ```

  Sets the starting boundary point for this **ccBoundaryTrackerRunParams**. This **ccBoundaryTrackerRunParams** must have been constructed using the track-only mode constructor.

  Parameters

  firstBoundaryPoint
  The starting point, specified in the client coordinate system.

  Throws

  ccBoundaryTrackerDefs::UnintializedData,
  This object was constructed with search-and track interface.
  ```

Notes

Call this function to specify a new starting point for track-only mode boundary tracking.

ccCADFile

```
#include <ch_cvl/cadfile.h>

class ccCADFile;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Behavior

This class encapsulates the conversion of a DXF format CAD file into a **ccGeneralShapeTree**. Currently only a subset of Autodesk, Inc.'s standard ASCII DXF file format is supported. Binary DXF files are not supported. DXF files with format versions not supported result in a throw from the converter.

Returned shape trees can represent the entire file, or specific layers or groups within the file. The tree representing the entire file is organized such that each layer of the drawing is represented as a **ccGeneralShapeTree** on a separate branch of the returned parent tree.

A **ccGeneralShapeTree** representing a particular layer or group will itself contain the various primitive shapes and custom objects within that layer or group. Each **ccShape** representing a custom object will itself be a **ccGeneralShapeTree** pointer object containing the set of primitive shapes and custom objects within the custom object. If a custom object is repeated more than once within the DXF file, it will result in the creation of a new **ccGeneralShapeTree** (the object will be replicated, not shared).

Transformations (including handedness information and coordinate system information) are supported by mapping all the supported primitive shapes into a single shape coordinate system, and are not explicitly represented. Primitives that are not supported are simply ignored.

DXF Versions Supported

Currently only a subset of the Autodesk standard ASCII DXF file format is supported. **ccCADFile** supports DXF formats corresponding to AutoCAD versions R12, R13, R14, and 2000. DXF versions corresponding to AutoCAD versions R11, 2000i, and 2002 should also work, but are not tested.

DXF File Organization

DXF files are organized into sections. The following table identifies these sections and shows what information is parsed from each one.

Sections	Contents
HEADER	Units, handedness, and base angle of the drawing.
TABLES	Layer information.
BLOCKS	Custom object name and definition.
ENTITIES	Visible objects that make up the basic geometric shapes of the drawing, as listed in the following table.
CLASSES	Application-specific, non-geometric information, not used by **ccCADFile**.
OBJECTS	Information about groups.

The following DXF entities are recognized by **ccCADFile** and are converted into the **ccShape**(s) shown.

DXF Entity	ccShape
ARC	**ccEllipseArc2**
CIRCLE	**ccEllipse2**
ELLIPSE	**ccEllipseArc2**
INSERT	**ccGeneralShapeTree**
LINE	**ccLineSeg**
LWPOLYLINE	**ccPolyline** or **ccCountourTree** (see notes below)
MLINE	**ccPolyline**
POLYLINE	**ccPolyline** or **ccCountourTree** (see notes below)
RAY	**ccLine**
SOLID	**ccPolyline**
SPLINE	**ccDeBoorSpline**
POINT	**cc2Point** (any Z position information is ignored)
XLINE	**ccLine**

The following DXF entity information is ignored:

- DXF entities not in the table above
- All three dimensional information
- Fill and style information
- Rounded corner and spline representations using POLYLINE

POLYLINE and LWPOLYLINE entities are represented as either **ccPolyline** or **ccCountourTree** objects. If the DXF entity consists of straight lines, a **ccPolyline** object is returned. If the DXF object contains any arcs, a **ccCountourTree** object consisting of **ccEllipseArc2** and **ccLineSeg** objects is returned.

The INSERT entity provides a mechanism for displaying a transformed version of a custom object from the BLOCKS section. Its **ccShape** is a **ccGeneralShapeTree** whose children consist of the transformed **ccShape** versions of its defining entities (which can include other INSERTs or trees). If a BLOCK is used more than once, it is replicated into a new **ccGeneralShapeTree**.

Constructors/Destructors

ccCADFile

```
ccCADFile();

ccCADFile(
  const ccCvlString &fileName,
  c_Int32 mode = cmStd ios::in);

~ccCADFile();

ccCADFile(const ccCADFile& rhs);
```

- `ccCADFile();`

  Default constructor. Creates an empty, closed **ccCADFile** object.

- ```
  ccCADFile(
    const ccCvlString &fileName,
    c_Int32 mode = cmStd ios::in);
  ```

  Opens and loads *fileName* containing the specified DXF format text file.

  Parameters

fileName	The pathname of the DXF format text file.
mode	The operating mode. *ios::in* is currently the only supported mode.
  ```

Throws

ccCADFile::NotImplemented
If *mode* is not *ios::in*.

ccCADFile::CanNotOpenFile
If *fileName* cannot found or cannot be opened for any reason.

ccCADFile::CanNotLoadFile
If there is a parsing error while reading the file.

ccCADFile::FileFormatNotSupported
If *fileName* is not a DXF format file.

- `~ccCADFile();`

Destructor. Deletes this object and closes any open DXF file.

- `ccCADFile(const ccCADFile& rhs);`

Copy constructor. Initialize this object using *rhs*.

Parameters

rhs The source of the copy.

Enumerations

UnitTypes `enum UnitTypes`

This enumeration defines the DXF file unit IDs.

Values
eUknown = 0
eScientific = 1
eDecimal = 2
eEngineering = 3
eArchitectural = 4
eFractional = 5
eWindowsdesktop = 6

Operators

operator=

```
ccCADFile& operator=(const ccCADFile& rhs);
```

Assignment operator.

Parameters

rhs Source of the assignment.

Public Member Functions

open

```
void open(
  const ccCvlString &fileName,
  c_Int32 mode = cmStd ios::in);
```

Opens *fileName* and loads the DXF format text file.

Parameters

fileName The path to the DXF format file to open.

mode The operating mode. *ios::in* is currently the only supported mode.

Throws

ccCADFile::NotImplemented
If *mode* is not ios::in.

ccCADFile::CanNotOpenFile
If *fileName* cannot found or cannot be opened for any reason.

ccCADFile::CanNotLoadFile
If there is a parsing error while reading the file.

ccCADFile::FileFormatNotSupported
If *fileName* is not a DXF format file.

isOpen

```
bool isOpen() const;
```

Returns true if the specified DXF file is open; otherwise returns false.

close

```
void close();
```

Closes the currently open DXF file.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

getUnits

```
UnitTypes getUnits() const;
```

Returns the unit ID of the currently open DXF file.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

layerNames

```
void layerNames(
  cmStd vector<cccVlString> &names) const;
```

Returns a vector of the layer names in the currently open DXF file.

The returned names are in the same order that the layer subtrees are added to a general shape tree using the method **shapeTree()** described below. For example, *names[0]* contains the name of the layer corresponding to the first added subtree. The layer corresponding to *names[i]* can be accessed directly by passing *names[i]* or the integer index *i* as the first argument to **layerShapeTree()**.

Parameters

names A vector of layer names contained in the currently open DXF file.

ccCADFile::FileNotOpen
If there is no open DXF file.

groupNames

```
void groupNames(
  cmStd vector<cccVlString> &names) const;
```

Returns a vector of the group names in the currently open DXF fi e.

The group corresponding to *names[i]* can be accessed directly by passing *names[i]* or the integer index *i* as the first argument to **groupShapeTree()**.

Parameters

names Group names contained in the currently open DXF file.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

numLayers

```
c_Int32 numLayers() const;
```

Returns the number of layers in the currently open DXF file.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

numGroups

```
c_Int32 numGroups() const;
```

Returns the number of groups in the currently open DXF file.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

shapeTree

```
void shapeTree(ccGeneralShapeTree& tree) const;
```

Adds the contents of the currently open DXF file to *tree*. Each DXF file layer is added to a separate branch of the tree.

Parameters

tree The shape tree where the DXF file's contents are to be added.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

layerShapeTree

```
void layerShapeTree(c_Int32 layer,
 ccGeneralShapeTree& tree) const;

void layerShapeTree(const ccCvlString& name,
 ccGeneralShapeTree& tree) const;
```

Adds the contents of a specified layer of the open DXF file to *tree*, where the layer is specified by name or index. The index of a layer is the zero-based position of its name in the vector returned by **layerNames()**. The contents of the layer are added on separate branches of the passed *tree*.

-
```
void layerShapeTree(c_Int32 layer,
 ccGeneralShapeTree& tree) const;
```

Parameters

layer The index number of a layer in the vector of layer names returned by **layerNames()**.

tree The shape tree where the layer is to be added.

Throws

ccCADFile::FileNotOpen
If there is no open DXF file.

ccCADFile::BadIndex
If *layer* is invalid.

- ```
  void layerShapeTree(const ccCvlString& name,
    ccGeneralShapeTree& tree) const;
  ```

  Parameters

  name The name of the layer in the currently open DXF file to be added.

  tree The shape tree where the layer is to be added.

  Throws

  ccCADFile::FileNotOpen
  If there is no open DXF file.

  ccCADFile::BadIndex
  If *name* is invalid.

groupShapeTree

```
void groupShapeTree(
  c_Int32 group,
  ccGeneralShapeTree& tree) const;

void groupShapeTree(
  const ccCvlString& grpName,
  ccGeneralShapeTree& tree) const;
```

Adds the contents of a specified group of the currently open DXF file to *tree*. The contents of the group are added on separate branches of the passed *tree*.

A group is an AutoCAD feature that allows for an arbitrary collection of DXF entities on any layer to be grouped together for display or processing. The group must be specified and names within AutoCAD before the DXF file is saved.

The group can be specified by group name or index number. The index of a group is the zero-based position of its name in the vector returned by **groupNames()**.

- ```
  void groupShapeTree(
    c_Int32 group,
    ccGeneralShapeTree& tree) const;
  ```

  Parameters

  group The index number of a group in the vector of group names returned by **groupNames()**.

  tree The shape tree where the group is to be added.

  Throws

  ccCADFile::FileNotOpen
  If there is no open DXF file.

ccCADFile::BadIndex
If *group* is invalid.

- ```
  void groupShapeTree(
    const ccCvlString& grpName,
    ccGeneralShapeTree& tree) const;
  ```

  Parameters

  grpName The group name to be added from the currently open DXF file.

  tree The shape tree where the group is to be added.

  Throws

  ccCADFile::FileNotOpen
  If there is no open DXF file.

  ccCADFile::BadIndex
  If *grpName* is invalid.
  ```

ccCalib2ParamsExtrinsic

```
#include <ch_cvl/ccalib.h>

class ccCalib2ParamsExtrinsic;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class holds extrinsic camera calibration parameters (properties of the image acquisition system that are external to the camera). These are:

- Orientation of the physical coordinate system with respect to the camera coordinate system, specified by three angles.
- Translation of the origin of the 3-D physical coordinate system with respect to the camera coordinate system.

The extrinsic parameters describe the rigid 3-D transform between the camera coordinate system and the physical coordinates system. This transform maps a 3-D vector v to R * v + t where t is the 3-D translation and R is the 3x3 matrix of rotations through angles contained in the extrinsic parameters.

When you use multiple views for a calibration application (for example, to create a **cc2XformCalib2** object), each view has a unique set of extrinsic parameters.

Constructors/Destructors

ccCalib2ParamsExtrinsic

```
ccCalib2ParamsExtrinsic(
  const cc3AngleVect& orientation,
  const cc3Vect& translation);

ccCalib2ParamsExtrinsic();
```

- ```
  ccCalib2ParamsExtrinsic(
    const cc3AngleVect& orientation,
    const cc3Vect& translation);
  ```

  Constructs an object with the specified orientation and translation.
  ```

Parameters

orientation An object that holds the three rotation angles of the axes of a 3-D coordinate system.

translation The 3-D origin translation between the physical coordinate system and the image coordinate system.

- `ccCalib2ParamsExtrinsic();`

Default constructor. Constructs an object initialized with all rotation angles = 0 and translation = (0, 0, 1).

Public Member Functions

orientation

```
const cc3AngleVect& orientation() const;
void orientation(const cc3AngleVect& orientation);
```

- `const cc3AngleVect& orientation() const;`

Returns the orientation parameter stored in this object.

- `void orientation(const cc3AngleVect& orientation);`

Sets a new orientation parameter.

Parameters

orientation The new orientation parameter.

translation

```
const cc3Vect& translation() const;
void translation(const cc3Vect& translation);
```

- `const cc3Vect& translation() const;`

Returns the translation parameter stored in this object.

- `void translation(const cc3Vect& translation);`

Sets a new translation parameter.

Parameters

translation The new translation parameter.

ccCalib2ParamsIntrinsic

```
#include <ch_cvl/ccalib.h>

class ccCalib2ParamsIntrinsic;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class holds intrinsic camera calibration parameters (properties in an image acquisition system that are internal to the camera). The intrinsic parameters are:

- The x- and y-scale (also called focal lengths)
- Skew
- The x- and y-translation
- Coefficient of radial distortion

These parameters remain constant as long as the internal camera settings (camera itself, lens used on the camera, focal length setting, aperture setting, and so on) remain constant. Also, the intrinsic camera calibration parameters are expected to stay constant even if external settings vary such as orientation of object plane with respect to the camera, lighting conditions, and so on.

Constructors/Destructors

ccCalib2ParamsIntrinsic

```
ccCalib2ParamsIntrinsic(
  const cc2Vect& scale,
  double skew,
  const cc2Vect& translation,
  double k1);

ccCalib2ParamsIntrinsic();
```

- ```
  ccCalib2ParamsIntrinsic(
    const cc2Vect& scale,
    double skew,
    const cc2Vect& translation,
    double k1);
  ```

  Constructs an intrinsic calibration parameters object from x and y scales, skew, x and y translation, and the radial distortion coefficient.

  Parameters

  scale — The x- and y-scale. For example, (.9, 1.1) for an x-scale factor of .9 and a y-scale factor of 1.1.

  skew — Equals $-S_y(\tan\theta)$, where S_y is the y-scale (see above) and θ is the shear angle as described in the **cc2Matrix** reference page.

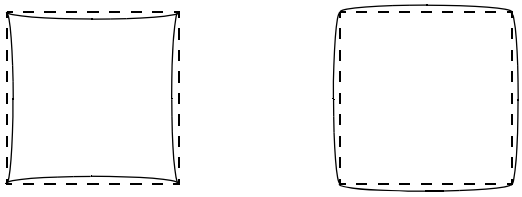
  translation — The x- and y-translation in client coordinates.

  k1 — Radial distortion coefficient. There are two types of radial distortion:

  1. Barrel distortion: This distortion bends the edges of a square outward from its center. In this case, k1 < 0.

  2. Pincushion distortion: This distortion bends the edges of a square inward towards its center. In this case, k1 > 0.
  ```

When *k1* = 0 there is no radial distortion. See the following diagram.



Pincushion distortion
k1 > 0

Barrel distortion
k1 < 1

Throws

cc2XformCalibDefs::BadParams
If **scale.x()** <= 0 or **scale.y()** <= 0.

- `ccCalib2ParamsIntrinsic();`

  Constructs an object initialized to its default state with *scale* = (1,1), *skew* = 0, *translation* = (0,0), and *k1* = 0.

Public Member Functions

scale

```
cc2Vect scale() const;
void scale(const cc2Vect& scale);
```

- `cc2Vect scale() const;`

  Returns the x- and y-scales stored in this object.

- `void scale(const cc2Vect& scale);`

  Sets new x- and y-scales.

  Parameters

  scale The new x- and y-scales.

  Throws

  cc2XformCalibDefs::BadParams
  If **scale.x()** <= 0 or **scale.y()** <= 0.

skew

```
double skew() const;
void skew(double skew);
```

See *skew* description in the constructor.

- `double skew() const;`

  Returns the skew stored in this object.

- `void skew(double skew);`

  Sets a new skew value.

  Parameters

  skew The new skew.

translation

```
cc2Vect translation() const;
void translation(const cc2Vect& trans);
```

The x- and y-translation in client coordinates.

- `cc2Vect translation() const;`

  Returns the x and y translation stored in this object.

- `void translation(const cc2Vect& trans);`

  Sets new x and y translation.

  Parameters

  trans The new x and y translation.

k1

```
double k1() const;
void k1(double k1);
```

See *k1* description in the constructor.

- `double k1() const;`

  Returns the radial distortion coefficient stored in this object.

- `void k1(double k1);`

  Sets a new radial distortion coefficient.

  Parameters

k1	The new radial distortion coefficient.

ccCalib2ParamsIntrinsic

ccCalib2VertexFeatureDefs

```
#include <ch_cvl/calibftr.h>

class ccCalib2VertexFeatureDefs;
```

Name space for calibration feature extraction related enums.

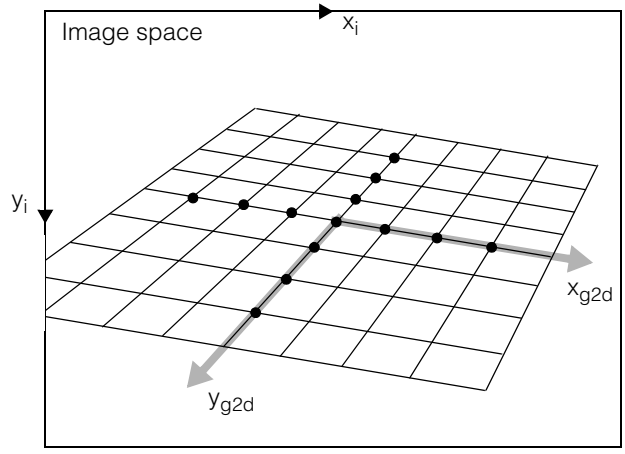
Enumerations

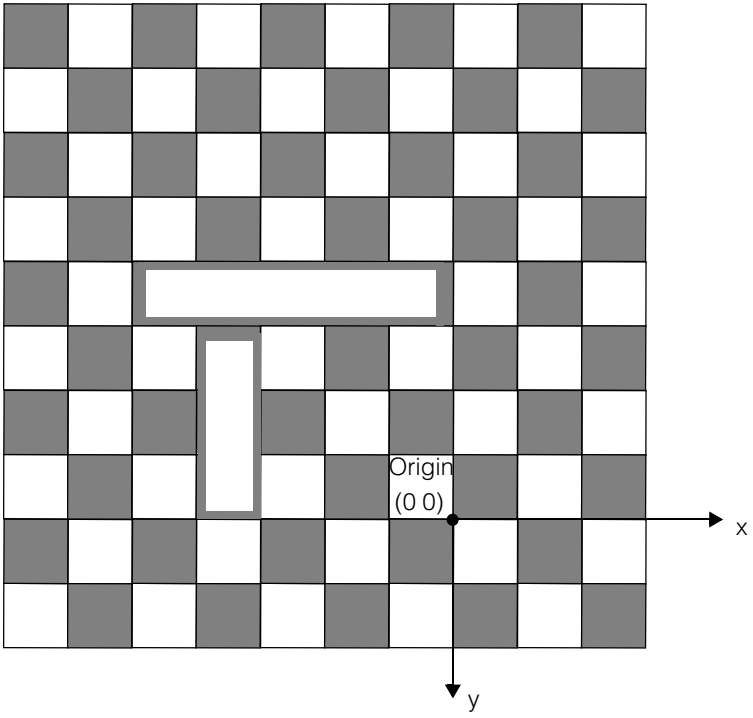
CorrespondMethod

```
enum CorrespondMethod;
```

This enumeration defines the modes used to set the origin and coordinate axes of the calibration plate Grid2D coordinate system. The correspondence method is also known as the *label mode* or *label method*. These enum modes are passed to **ccCalib2VertexFeatureParams::labelMode(***mode***)**.

If it is crucial to have the Grid2D coordinate system stationary with respect to the physical calibration plate (for example, in multiple view calibration), you should only use the *eUseFiducial* mode.

Mode	Meaning
eNone = 0	The origin of the Grid2D coordinate system is set at the vertex closest to the center of the image. The Grid2D x-axis is set on the row or column of vertices that passes through the origin, and is most closely aligned with the image space x-axis. The image space x-axis and the Grid2D x-axis, as viewed in the image, point in approximately the same direction. See the following figure.  Keep in mind that the checkerboard in the image can be at any oblique angle depending on the camera angle, and it can be distorted. This is the default mode.
eUseOrigin	Specifies that the vertex closest to a designated location be used as the origin of the Grid2D coordinate system. The designated origin can be set using **ccCalibrationVertexFeatureParams::origin()**. The direction of the axes is set exactly as in *eNone* mode above.

Mode	Meaning
eUseFiducial	Specifies that the use of a Cognex-specific fiducial target be found in the image and used to set the Grid2D coordinate system. The origin (0,0) and the axes (x and y) of the Grid2D coordinate system are chosen as shown in the following diagram of a calibration plate containing a fiducial marker. The Cognex part number for these calibration plates is 320-0015. If you use your own calibration plate, it must conform to specifications contained in an engineering drawing available from Cognex. This specification allows for any physical size for checkers but once physical checker size is decided, the width, length of the fiducial bars, polarities of checkers, and the fiducial must be according to specification. Note that size of all checkers in acquired images of a physical calibration plate must be at least 15x15 pixels. The following is an example calibration plate with a fiducial mark. 

Mode	Meaning
eUse DataMatrix	Specifies that standard Data Matrix codes are to be used as calibration fiducials that are uniquely identifiable in the calibration target. Each Data Matrix code must encode *either* its physical position – Version 0 format – *or* its physical position and the grid pitch – Version 1 format. This mode will ignore any information about the physical grid pitch that may be encoded in the Data Matrix codes, and will instead use the value **ccCalib2VertexFeatureParams::physicalGridPitch()** specified in the input parameters. See the API for the CVL symbol tool <*ch_cvl/acusymbl.h*> and the CVL Vision Tools Guide for more details about Data Matrix codes and any associated terminology used in the rest of this document. Also see the CVL Vision Tools Guide for the requirements this mode imposes on all Data Matrix codes in the image.
eUse DataMatrix WithGridPitch	Similar to *eUseDataMatrix*, but in this mode each Data Matrix code *must* encode its data in the Version 1 format only. In this mode, the tool will ignore the value **ccCalib2VertexFeatureParams::physicalGridPitch()** in the input parameters, and will instead use the value of physical grid pitch encoded in the Data Matrix codes. If the encoded value of the physical grid pitch is less than or equal to zero, the tool will throw *ccCalib2VertexFeatureDefs::DataMatrixParseError*

`enum`

This enumeration defines default calibration plate parameters.

Value	Meaning
kDefaultGridPitch = 1	Default grid pitch in client units.

Algorithm

```
enum Algorithm;
```

This enumeration defines the algorithms used to extract and label features (tile vertices) from the image.

Value	Meaning
eStandard = 0	Appropriate for clean images with minimal noise, contrast variations, or image defects.
eExhaustive = 1	An exhaustive algorithm is used to locate and label the tile vertices. This algorithm is significantly more robust than *eStandard*, although it may take longer to run. Cognex recommends the use of *eExhaustive* in all cases. Note, however, that the vertex locations returned by *eExhaustive* may differ slightly from those returned by *eStandard*.

Value	Meaning
eExhaustiveMultiRegion = 2	Similar to *eExhaustive*, but it returns feature correspondences from all contiguous regions of checkerboard features in which a Data Matrix was successfully decoded (unlike *eStandard* or *eExhaustive*). It can only be used with Data Matrix fiducials (that is, when label mode is *eUseDataMatrix* or *eUseDataMatrixWithGridPitch*). *eExhaustiveMultiRegion* runs slower than *eExhaustive*. However, if robustness is more important than speed, then it is recommended to use *eExhaustiveMultiRegion* instead of *eExhaustive* when Data Matrix fiducials are employed, for the following reason. Sometimes a portion of the calibration target may be obscured due to physical obstructions or poor lighting, causing the checkerboard grid to get split into distinct regions of contiguous features. When Data Matrix codes are used as fiducials, it becomes possible to extract features from these multiple regions as long as each region of contiguous features has a Data Matrix code embedded inside. This is possible because each Data Matrix code encodes its position in the grid. Also note that this mode returns features from maximally connected sub-grids. It does not intentionally separate connected features into different regions. So, when the features are fully connected, *eExhaustiveMultiRegion* returns exactly the same result as *eExhaustive*. When the image includes disconnected regions of features, *eExhaustiveMultiRegion* may return more features than *eExhaustive* because *eExhaustive* only returns features from one fully connected region and *eExhaustiveMultiRegion* can return features from multiple connected regions.
kDefaultAlgorithm = eStandard	The default algorithm (to preserve backwards compatibility.

ccCalib2VertexFeatureDefs

ccCalib2VertexFeatureParams

```
#include <ch_cvl/calibftr.h>

class ccCalib2VertexFeatureParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class encapsulates all the parameters required to configure feature extraction from a checkerboard type calibration plate image.

Constructors/Destructors

ccCalib2VertexFeatureParams

```
ccCalib2VertexFeatureParams();

ccCalib2VertexFeatureParams(
  const cc2Vect& gridPitch,
  ccCalib2VertexFeatureDefs::CorrespondMethod mode =
                          ccCalib2VertexFeatureDefs::eNone);
```

- `ccCalib2VertexFeatureParams();`

  Default constructor. Default values are:

  gridPitch = (1, 1)
  mode = *eNone*

- ```
  ccCalib2VertexFeatureParams(
    const cc2Vect& gridPitch,
    ccCalib2VertexFeatureDefs::CorrespondMethod mode =
                            ccCalib2VertexFeatureDefs::eNone);
  ```

  Constructs a vertex feature parameters object using the given parameters.

  Parameters

gridPitch	The calibration plate grid pitch.
mode	The correspondence mode.
  ```

Public Member Functions

physicalGridPitch

```
void physicalGridPitch(const cc2Vect& gridPitch);

const cc2Vect& physicalGridPitch() const;
```

Specifies physical units along the x- and y-axes of the Plate2D coordinate system. The x-component of *gridPitch* is the distance between any two adjacent checker vertices when the line joining them is parallel to the Plate2D x-axis. The y-component of *gridPitch* is the distance between any two adjacent checker vertices when the line joining them is parallel to the Plate2D y-axis.

The default physical grid pitch is (1, 1).

- `void physicalGridPitch(const cc2Vect& gridPitch);`

  Sets a new grid pitch.

  Parameters

  gridPitch The new grid pitch.

  Throws

  ccCalib2VertexFeatureDefs::BadParam
  If either component of *gridPitch* is less than or equal to zero.

- `const cc2Vect& physicalGridPitch() const;`

  Returns the current grid pitch.

origin

```
void origin(const cc2Vect& o);

const cc2Vect& origin() const;
```

An origin in client units to be used for labeling of returned feature points. The vertex closest to this point will be used as the origin for point correspondence when operating in *ccCalib2VertexFeatureDefs::eUseOrigin* mode. The default is (0, 0).

- `void origin(const cc2Vect& o);`

  Sets a new origin.

  Parameters

  o The new origin.

- `const cc2Vect& origin() const;`

  Returns the current origin.

labelMode

```
void labelMode(
  ccCalib2VertexFeatureDefs::CorrespondMethod m);

ccCalib2VertexFeatureDefs::CorrespondMethod labelMode()
                                                    const;
```

The method used for labeling detected vertex points in the calibration plate image. The default is *ccCalib2VertexFeatureDefs::eNone.*

- `void labelMode(`
  `ccCalib2VertexFeatureDefs::CorrespondMethod m);`

  Sets a new label mode.

  Parameters

  m The new label mode.

  Throws

  ccCalib2VertexFeatureDefs::BadParam
  If *m* is not a valid entry from the *ccCalib2VertexFeatureDefs::CorrespondMethod* enum

- `ccCalib2VertexFeatureDefs::CorrespondMethod labelMode()`
  `const;`

  Returns the current label mode.

algorithm

```
void algorithm(
  ccCalib2VertexFeatureDefs::Algorithm algorithm);

ccCalib2VertexFeatureDefs::Algorithm algorithm()const;
```

The algorithm to use when extracting and labeling vertices. The default is *ccCalib2VertexFeatureDefs::eStandard.* Cognex recommends the use of *ccCalib2VertexFeatureDefs::eExhaustive*, which is more robust, particularly in cases of low or uneven contrast, image blur, reflections, or plate defects.

- `void algorithm(`
  `ccCalib2VertexFeatureDefs::Algorithm algorithm);`

  Sets the algorithm.

Parameters

m The algorithm.

Throws

ccCalib2VertexFeatureDefs::BadParam
If the input is not a valid choice from the enum *ccCalib2VertexFeatureDefs::Algorithm*

- `ccCalib2VertexFeatureDefs::Algorithm algorithm()const;`

Returns the current algorithm.

dataMatrixTimeoutSeconds

```
void dataMatrixTimeoutSeconds(double timeoutSec);

double dataMatrixTimeoutSeconds() const;
```

- `void dataMatrixTimeoutSeconds(double timeoutSec);`

Sets the timeout in seconds to find a single Data Matrix code in the image (or a region in the image if algorithm is *eExhaustiveMultiRegion*). The tool will only spend the specified amount of seconds to find and decode each Data Matrix code.

Parameters

timeoutSec The timeout value in seconds to set.

Notes

This parameter is used only when *labelMode* is *eUseDataMatrix* or *eUseDataMatrixWithGridPitch*.

If the algorithm is *eExhaustiveMultiRegion* and the tool cannot find and decode a Data Matrix code inside a region of checkerboard features within the specified time, it will not include any features from that region.

Throws

ccCalib2VertexFeatureDefs::BadParam
If *timeoutSec* is less than or equal to 0.

The default is *HUGE_VAL*

- `double dataMatrixTimeoutSeconds() const;`

Gets the timeout.

Deprecated Members

The following constructor and functions are deprecated and are maintained here for backward compatibility only.

ccCalib2VertexFeatureParams

```
ccCalib2VertexFeatureParams(
  const cc2Vect& gridPitch,
  const cc2Vect& tileSize =
       cc2Vect(ccCalib2VertexFeatureDefs::kDefaultTileSize,
       ccCalib2VertexFeatureDefs::kDefaultTileSize),
  ccCalib2VertexFeatureDefs::CorrespondMethod mode =
                          ccCalib2VertexFeatureDefs::eNone);
```

nominalTileSize

```
void nominalTileSize(const cc2Vect& tileSize);

const cc2Vect& nominalTileSize();
```

ccCalib2VertexFeatureParams

ccCalib2VertexFeatureParseResult

```
#include <ch_cvl/calibftr.h>

class ccCalib2VertexFeatureParseResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class contains the result of parsing the string that was encoded in a Data Matrix code.

Constructors/Destructors

ccCalib2VertexFeatureParseResult

```
ccCalib2VertexFeatureParseResult();
```

Constructs the object using the default values.

Notes

Compiler generated destructor, copy constructor, and assignment operator are used.

Operators

operator==

```
bool operator==(
  const ccCalib2VertexFeatureParseResult& rhs) const;
```

Equality test operator. Returns true if *this is the same as rhs, or false otherwise.

Parameters

rhs	The object to compare with this object.

operator!=

```
bool operator!=(
  const ccCalib2VertexFeatureParseResult& rhs) const;
```

Inequality test operator. Returns false if *this is the same as rhs, or true otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

isComputed `bool isComputed() const;`

Returns a Boolean flag that indicates if this result object was computed or not.

The default is false.

Notes

It is set to true when **set()** is called.

hasPositionGrid2D

`bool hasPositionGrid2D() const;`

Returns whether or not this result has a valid **positionGrid2D()**.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

positionGrid2D `const cc2Vect& positionGrid2D() const;`

Returns the (X,Y) Grid2D position of the Data Matrix code.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false
or
hasPositionGrid2D() is false.

hasGridPitch `bool hasGridPitch() const;`

Returns whether or not this result has a valid **gridPitch()**.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

gridPitch `const cc2Vect& gridPitch() const;`

Returns the grid pitch of the checkerboard tiles along the X and Y axes of Plate2D in physical measurement units.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false
or
hasGridPitch() is false.

hasUnitString

```
bool hasUnitString() const;
```

Returns whether or not this result has a valid **unitString()**.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

unitString

```
const ccCvlString& unitString() const;
```

Returns the measurement units in which *gridPitch* was specified.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false
or
hasUnitString() is false.

ccCalib2VertexFeatureParseResult

ccCalib2VertexFeatureResult

```
#include <ch_cvl/calibftr.h>

class ccCalib2VertexFeatureResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class contains the result of the calibration.

Constructors/Destructors

ccCalib2VertexFeatureResult

```
ccCalib2VertexFeatureResult() : isComputed_(false);
```

Constructs a default object.

Notes

Compiler generated destructor, copy constructor, and assignment operator are used.

Operators

operator==

```
bool operator==(
  const ccCalib2VertexFeatureResult& rhs) const;
```

Equality test operator. Returns true if *this is the same as rhs, or false otherwise.

Parameters

rhs	The object to compare with this object.

operator!=

```
bool operator!=(
  const ccCalib2VertexFeatureResult& rhs) const;
```

Inequality test operator. Returns false if *this is the same as rhs, or true otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns a Boolean flag that indicates if this result object was computed or not.

The default is false.

client2DPlate2DPairs

```
const ccCrspPairVector& client2DPlate2DPairs() const;
```

Returns a vector of correspondence pairs. Each correspondence pair contains the Client2D position of an extracted vertex point and its corresponding Plate2D position.

Notes

If label mode is *eExhaustiveMultiRegion*, multiple regions of checkerboard features may be detected. This vector will contain feature correspondences from all grids of features that were detected.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

client2DPlate2DPairsMultiRegion

```
const cmStd vector<ccCrspPairVector>&
client2DPlate2DPairsMultiRegion() const;
```

Returns a vector of ccCrspPairVectors, where each ccCrspPairVector contains feature correspondences from a single region of contiguous checkerboard features.

Notes

If label mode is not *eExhaustiveMultiRegion*, features from only one region of checkerboard features will have been detected and this vector will contain only one element.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

symbolInfo

```
const cmStd vector<ccCalib2VertexFeatureSymbolInfo>&
symbolInfo() const;
```

Returns a vector of objects that contain the data decoded from each Data Matrix that was successfully found, decoded, and parsed.

Notes

If the tool is **not** running in *eUseDataMatrix* mode or *eUseDataMatrixWithGridPitch* mode, this vector will be empty.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

ccCalib2VertexFeatureResult

ccCalib2VertexFeatureSymbolInfo

```
#include <ch_cvl/calibftr.h>

class ccCalib2VertexFeatureSymbolInfo;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class contains data decoded from the Data Matrix code.

Constructors/Destructors

ccCalib2VertexFeatureSymbolInfo

```
ccCalib2VertexFeatureSymbolInfo() : isComputed_(false);
```

Constructs a default object.

Notes

Compiler generated destructor, copy constructor, and assignment operator are used.

Operators

operator==

```
bool operator==(
  const ccCalib2VertexFeatureSymbolInfo& rhs) const;
```

Equality test operator. Returns true if *this is the same as rhs, or false otherwise.

Parameters

rhs	The object to compare with this object.

operator!=

```
bool operator!=(
  const ccCalib2VertexFeatureSymbolInfo& rhs) const;
```

Inequality test operator. Returns false if *this is the same as rhs, or true otherwise.

Parameters

rhs	The object to compare with this object.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns a Boolean flag that indicates if this result object was computed or not.

The default is false.

parseResult

```
const ccCalib2VertexFeatureParseResult& parseResult()
const;
```

Returns the result of parsing the data encoded in the Data Matrix code.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

dataMatrixResult

```
const ccAcuSymbolResult& dataMatrixResult() const;
```

Returns the found and decoded Data Matrix code.

Throws

ccCalib2VertexFeatureDefs::NotComputed
If **isComputed()** is false.

ccCalibDefs

```
#include <ch_cvl/calib.h>

class ccCalibDefs;
```

A name space that holds enumerations and constants used with the Calibration tool classes.

Enumerations

CalibType

```
enum CalibType;
```

This enumeration defines the types of calibration supported by the Calibration tool.

Value	Meaning
eGridLinear	Linear calibration using a grid of dots.
eGridPolynomialOrder3	Third-order polynomial calibration using a grid of dots.
eGridPolynomialOrder5	Fifth-order polynomial calibration using a grid of dots.

Polarity

```
enum Polarity;
```

This enumeration defines the polarity of the calibration grid image used to perform the calibration.

Value	Meaning
eDarkOnLight	The image consists of dark dots on a light background.
eLightOnDark	The image consists of light dots on a dark background.
eAutoDetect	The Calibration tool automatically determines the polarity of the image.

ccCalibDefs

ccCalibrateLineScanCameraDefs

```
#include <ch_cvl/ccalibls.h>

class ccCalibrateLineScanCameraDefs;
```

Namespace for enumerations related to line scan calibration.

Enumerations

CalibrationYAxisAdjustmentMode

```
enum CalibrationYAxisAdjustmentMode;
```

This enumeration defines how the y-axis of calibrated space is established during line scan calibration.

Value	Meaning
eUsePlateOrientation	The y-axis is defined by the orientation of the fiducial mark on the calibration plate.
eAdjustOrientationToScanDirection	The y-axis is defined by the apparent motion of the calibration plate as the line scan image is acquired. To use the 1D image unwarp tool to generate a rectified image from a calibrated line scan camera, you must specify this adjustment mode.
kDefaultCalibrationYAxisAdjustmentMode	The default adjustment mode (*eUsePlateOrientation*).

ccCalibrateLineScanCameraDefs

ccCalibrateLineScanCameraParams

```
#include <ch_cvl/ccalibls.h>

class ccCalibrateLineScanCameraParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class encapsulates all the parameters required to configure the line scan camera calibration tool.

Constructors/Destructors

ccCalibrateLineScanCameraParams

```
ccCalibrateLineScanCameraParams();
```

Default-constructs an object with the following values:

- **useDistanceFromCameraToTarget**: false
- **distanceFromCameraToTarget**: 0.0
- **calibrationYAxisAdjustmentMode**: *kDefaultCalibrationYAxisAdjustmentModel*

Public Member Functions

useDistanceFromCameraToTarget

```
bool useDistanceFromCameraToTarget() const;

void useDistanceFromCameraToTarget(
   bool useDistanceFromCameraToTarget);
```

- ```
  bool useDistanceFromCameraToTarget() const;
  ```

  Returns true if this **ccCalibrateLineScanCameraParams** specifies that the supplied camera-to-target distance be used when calibrating the camera, false if the specified distance is ignored.
  ```

- ```
  void useDistanceFromCameraToTarget(
     bool useDistanceFromCameraToTarget);
  ```

  Specifies whether or not the distance specified for **distanceFromCameraToTarget()** is used when calibrating the camera. If you specify and use this distance, it must be accurate within 10%. Specifying the distance can improve the accuracy of the calibration.

  Parameters

  useDistanceFromCameraToTarget

  Specify true to use the supplied distance, false to ignore it.

distanceFromCameraToTarget

```
double distanceFromCameraToTarget() const;

void distanceFromCameraToTarget(
   double distanceFromCameraToTarget);
```

- ```
  double distanceFromCameraToTarget() const;
  ```

  Returns the currently specified camera-to-target distance for this **ccCalibrateLineScanCameraParams**. This value is only used if **useDistanceFromCameraToTarget()** is true.

- ```
  void distanceFromCameraToTarget(
     double distanceFromCameraToTarget);
  ```

  Sets the camera-to-target distance for this **ccCalibrateLineScanCameraParams**. This value is only used if **useDistanceFromCameraToTarget()** is true.

  If you specify a distance, it must be accurate to within 10% of the actual distance from the surface of the calibration plate and the surface of the image sensor in the camera. Specifying an accurate distance can improve the accuracy of the computed calibration.

  The distance must be specified in the same units in which you express the physical positions of the calibration plate vertices.

  Parameters

  distanceFromCameraToTarget

  The measured distance between the calibration plate surface and the camera image sensor surface. If the supplied value is 0, **cfCalibrateLineScanCamera()** will *ccCalibrateLineScanCameraDefs::BadParams*.
  ```

Throws

ccCalibrateLineScanCameraDefs::BadParams

distanceFromCameraToTarget is less than 0 or equal to HUGE_VAL.

calibrationYAxisAdjustmentMode

```
ccCalibrateLineScanCameraDefs::CalibrationYAxisAdjustmentMode
   calibrationYAxisAdjustmentMode() const:

void calibrationYAxisAdjustmentMode (
   ccCalibrateLineScanCameraDefs::CalibrationYAxisAdjustmentMode
   calibrationYAxisAdjustmentMode);
```

- `ccCalibrateLineScanCameraDefs::CalibrationYAxisAdjustmentMode calibrationYAxisAdjustmentMode() const:`

  Returns the y-axis adjustment mode specified for this object. The returned value is one of the following:

  ccCalibrateLineScanCameraDefs::eUsePlateOrientation
  ccCalibrateLineScanCameraDefs::eAdjustOrientationToScanDirection

- `ccCalibrateLineScanCameraDefs::CalibrationYAxisAdjustmentMode calibrationYAxisAdjustmentMode() const:`

  Sets the y-axis adjustment mode to be used when calibrating a line scan camera. If you specify *ccCalibrateLineScanCameraDefs::eUsePlateOrientation*, then the calibrated space y-axis orientation is defined by the orientation of the fiducial plate in the calibration image.

  If you specify *ccCalibrateLineScanCameraDefs::eAdjustOrientationToScanDirection*, then the calibrated space y-axis orientation is defined by the apparent motion of the plate during image acquisition.

  Parameters

  calibrationYAxisAdjustmentMode

  The mode to use. *calibrationYAxisAdjustmentMode* must be one of the following values:

  ccCalibrateLineScanCameraDefs::eUsePlateOrientation
  ccCalibrateLineScanCameraDefs::eAdjustOrientationToScanDirection

Notes

You must specify *ccCalibrateLineScanCameraDefs::eAdjustOrientationToScanDirection* in order to use the 1D warper to produce a rectified line scan image.

Operators

operator==

```
bool operator==(
   const ccCalibrateLineScanCameraParams &that) const;
```

Returns true if the supplied object is equal (all members are exactly equal) to this one.

Parameters

that	The object to compare.

ccCaliperBaseResultSet

```
#include <ch_cvl/caliper.h>

class ccCaliperBaseResultSet: public virtual ccPersistent;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	Complex

This base class contains some of the results returned by both correlation and edge mode.

Note You should not attempt to instantiate this class directly.

Constructors/Destructors

ccCaliperBaseResultSet

```
ccCaliperBaseResultSet();
```

Constructs **ccCaliperBaseResultSet** with no results.

Operators

operator==

```
bool operator== (const ccCaliperBaseResultSet& that) const;
```

Two **ccCaliperBaseResultSet** objects are equal if and only if the following items are all the same: the vectors of **ccResult**s; the vectors of **ccCaliperResultEdge**s; the caliper data; the user-specified origin in client coordinates; the unit vectors in the search direction; the run mode; the result angles; and the clipping status.

Public Member Functions

results

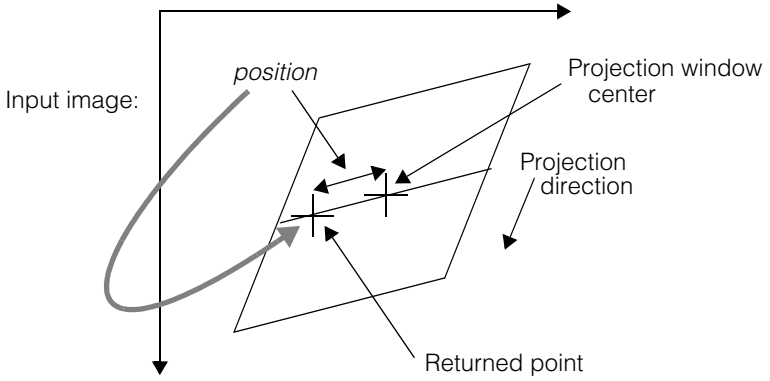
```
const cmStd vector<ccCaliperOneResult>& results () const;
```

Returns a vector of **ccCaliperOneResult**. Every edge or edge pair for which a score was computed is included in the vector returned by this function.

mapPositionToPoint

```
cc2Vect mapPositionToPoint (double position) const;
```

Map the given position within the projection region to a point in the client coordinates of the input image. If the projection image was constructed by the Caliper tool, the position is specified relative to the center of the projection region and the returned point lies on the center line of the projection region in the input image, as shown in the following figure:



If you supplied the projection image to the Caliper tool, then this function returns the two-dimensional point within the input image specified by the distance from the origin that you supplied.

Parameters

position — The position of interest, relative to either the center of the projection region or to the origin that you supplied with the projection image.

Notes

This function can be used to convert positions to 2-D client coordinates.

projectTime

```
double projectTime () const;
```

Returns the amount of time, in seconds, required to create the projection image.

filterTime

```
double filterTime () const;
```

Returns the amount of time, in seconds, required to create the filtered image.

scoreTime

```
double scoreTime () const;
```

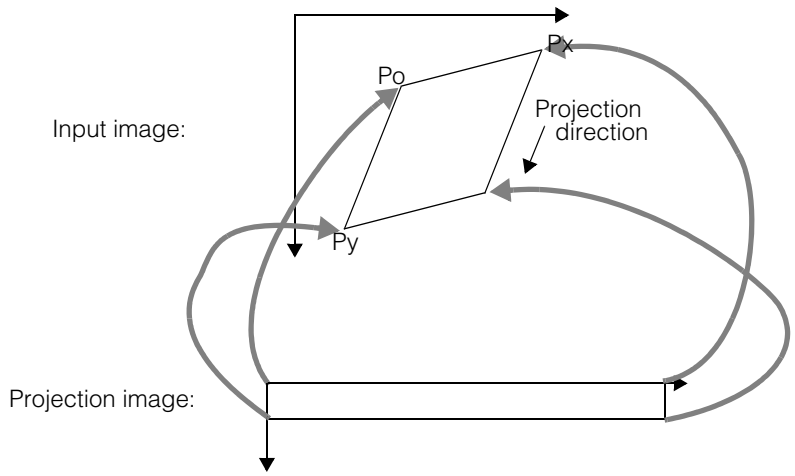
Returns the amount of time, in seconds, required to compute the scores for the edges in the input image.

projectedImage

```
ccPelBuffer_const<c_UInt32> projectedImage () const;
```

Returns the projection image. This image was created by projecting the pixels within the caliper window.

The **cc2Xform** returned by calling **projectedImage()::clientFromImageXform()** maps two-dimensional points within the projection image to the corresponding points within the input image client coordinate system, as shown in the following figure:



weightsImage

```
ccPelBuffer_const<c_UInt32> weightsImage () const;
```

Return the weights image. If clipping occurred, each pixel in this image contains the number of pixels that where summed together to produce the corresponding pixel in the projected image.

To determine if clipping occurred, call the **clipped()** member function.

The image returned by this function is only valid if clipping occurred and if auto-clipping mode is enabled.

filteredImage

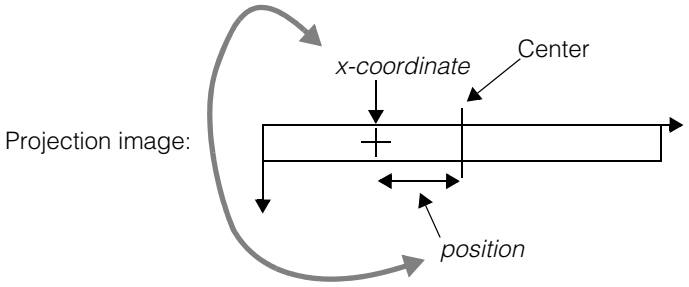
```
ccPelBuffer_const<double> filteredImage () const;
```

Return the filtered projection image. The width of the image returned by this function is smaller than that returned by **projectedImage()**, but the client coordinate system of the returned image lets you convert points between the projection image and the filtered image.

projectedXFromPosition

```
cc1Xform projectedXFromPosition() const;
```

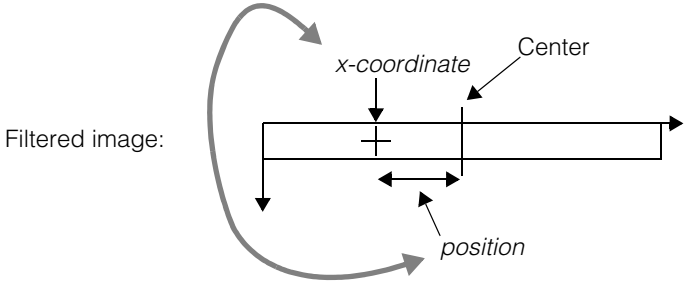
Returns a **cc1Xform** that can be used to map back and forth between a one-dimensional *position* and the corresponding *x-coordinate* in the projection image, as shown in the following figure:



filteredXFromPosition

```
cc1Xform filteredXFromPosition() const;
```

Returns a **cc1Xform** that can be used to map back and forth between a one-dimensional *position* and the corresponding *x-coordinate* in the filtered image, as shown in the following figure:



resultAngle

```
ccRadian resultAngle() const;
```

Returns the angle at which the Caliper tool was applied to generate this **ccCaliperBaseResultSet**.

clipped

```
bool clipped() const;
```

Returns true if the source image was clipped by the projection region, false if no clipping occurred.

ccCaliperBaseResultSet

ccCaliperBaseRunParams

```
#include <ch_cvl/caliper.h>

class ccCaliperBaseRunParams: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The **ccCaliperBaseRunParams** class is a base class from which the following two run-time parameter classes are derived:

- **ccCaliperRunParams**, which includes run-time parameters for edge mode
- **ccCaliperCorrelationRunParams**, which includes run-time parameters for correlation mode

The **ccCaliperBaseRunParams** class includes run-time parameters which are common to both edge and correlation mode.

Note You should not attempt to instantiate this class directly.

Enumerations

Timer

```
enum Timer;
```

This enumeration specifies the type of timer used to compute times in the result with the caliper tool.

Value	Meaning
eElapsedTime	Use the elapsed clock time in seconds. Produces accurate results on all machine types, but the overhead of computing the elapsed time can be noticeable (several microseconds) when using a large number of small (fast) calipers.
eScaledProcessorCycles	Use the number of processor cycles (scaled to seconds). Does not always produce accurate results on machines with a variable clock rate, but has a much lower overhead. Some machines (especially laptops) may use a variable clock rate to conserve power.

Public Member Functions

scanParams

```
const ccCaliperScanParams& scanParams() const;

void scanParams(ccRadian start, ccRadian end,
  ccRadian increment, bool interpolate);
```

- `const ccCaliperScanParams& scanParams() const;`

  Returns the **ccCaliperScanParams** object associated with this **ccCaliperBaseRunParams**.

- `void scanParams(ccRadian start, ccRadian end, ccRadian increment, bool interpolate);`

  Sets the parameters of the **ccCaliperScanParams** object associated with this **ccCaliperBaseRunParams**.

  Parameters

  start — The start of the range of angles to be scanned in radians. The angle is from the client coordinate system x-axis to the projection direction.

end The end of the range of angles to be scanned in radians. The angle is from the client coordinate system x-axis to the projection direction.

increment The amount by which to increment the angle between scans.

interpolate If true, intermediate angles are evaluated.

scanEnable

```
bool scanEnable() const;
void scanEnable(bool on);
```

- `bool scanEnable() const;`

  Returns true if scan mode is enabled, false if it is disabled.

- `void scanEnable(bool on);`

  Enables or disables scan mode. If scan mode is enabled, the caliper is successively applied at the angles specified in the **ccCaliperScanParams**. The angle which yields the highest average score for all results is used to compute the score for the tool.

  Parameters

  on If true, scan mode is enabled. If false, scan mode is disabled.

autoClip

```
bool autoClip() const;
void autoClip(bool state);
```

- `bool autoClip() const;`

  Returns true if auto-clipping mode is enabled, false if it is disabled.

- `void autoClip(bool state);`

  Enables or disables auto-clipping mode. If auto-clipping mode is enabled, if the projection region is clipped by the input image, the projection image is automatically resized to include only the pixels which were not affected by clipping. If auto-clipping mode is disabled, the tool throws an error if the projection region is clipped by the input image.

  Parameters

  state If true, auto-clipping mode is enabled. If false, auto-clipping mode is disabled.

Notes

Enabling auto-clipping mode will cause the **cfCaliperRun()** function to run as much as 10 times more slowly than when auto-clipping mode is disabled.

computeIntermediateTimes

```
bool computeIntermediateTimes() const;
void computeIntermediateTimes(bool val);
```

- `bool computeIntermediateTimes() const;`

  Returns true if the caliper tool should compute intermediate times for projection, filtering, scoring, and edge or peak detection, false otherwise. If false, the getters for these times will return zero.

- `void computeIntermediateTimes(bool val);`

  Sets whether the caliper tool should compute intermediate times for projection, filtering, scoring, and edge or peak detection. If false, the getters for these times will return zero.

  Parameters

  val — Set to true if the tool should calculate intermediate results, false otherwise. The default is true.

Note Querying the system timer with *QueryPerformanceCounter* costs, for example, ~1.7 microseconds on a 650 MHz system. Computing intermediate times can thus add nontrivial overhead for very small calipers. To get the fastest performance, turn timers off.

timerType

```
Timer timerType() const;
void timerType(Timer t);
```

- `Timer timerType() const;`

  Retrieves the type of timer used to compute times in the result.

- `void timerType(Timer t);`

  Sets the type of timer used to compute times in the result.

Parameters

t	The type of timer. Must be one of: *ccCaliperBaseRunParams::eElapsedTime* *ccCaliperBaseRunParams::eScaledProcessorCycles* (default)

Note For fastest execution of multiple calipers, set calipers once in the application code.

ccCaliperBaseRunParams

ccCaliperCircleFinderAutoRunParams

```
#include <ch_cvl/clpfind.h>

class ccCaliperCircleFinderAutoRunParams :
                        public ccCaliperFinderBaseAutoRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Encapsulates the Circle Finder tool run-time parameters for running the tool in *auto* mode.

Constructors/Destructors

ccCaliperCircleFinderAutoRunParams

```
ccCaliperCircleFinderAutoRunParams (
  const ccCircle &expectedCircle = ccCircle(),
  c_Int16 numCalipers = 0,
  const ccDPair &caliperSize = ccDPair(0, 0),
  const ccCaliperRunParams &caliperRunParams =
                                      ccCaliperRunParams(),
  const ccDPair &caliperSampling = ccDPair(1, 1),
  const ccAngleRange &angleRange =
                              ccAngleRange::FullAngleRange(),
  ccAffineSamplingParams::Interpolation
                         interpolationMethod =
                         ccAffineSamplingParams::eBilinear,
  const ccCircleFitParams &circleFitParams =
                                       ccCircleFitParams(),
  bool centrifugal = true,
  const cc2Xform& startPose = cc2Xform(),
  bool decrementNumIgnore = true,
  bool placeCalipersSymmetrically = true);

virtual ~ccCaliperCircleFinderAutoRunParams() {}
```

-
```
ccCaliperCircleFinderAutoRunParams (
  const ccCircle &expectedCircle = ccCircle(),
  c_Int16 numCalipers = 0,
  const ccDPair &caliperSize = ccDPair(0, 0),
```

```
const ccCaliperRunParams &caliperRunParams =
                                        ccCaliperRunParams(),
const ccDPair &caliperSampling = ccDPair(1, 1),
const ccAngleRange &angleRange =
                              ccAngleRange::FullAngleRange(),
ccAffineSamplingParams::Interpolation
                        interpolationMethod =
                        ccAffineSamplingParams::eBilinear,
const ccCircleFitParams &circleFitParams =
                                         ccCircleFitParams(),
bool centrifugal = true,
const cc2Xform& startPose = cc2Xform(),
bool decrementNumIgnore = true,
bool placeCalipersSymmetrically = true);
```

Constructs a Circle Finder tool run-time parameters object.

Notes

Default constructed objects do not contain valid run-time parameters.

Parameters

expectedCircle The circle you expect the tool to find in images.

numCalipers The number of calipers the tool should use to find the circle.

caliperSize The caliper size, width and height in client coordinates. All calipers are the same size.

caliperRunParams
The run-time parameters object to be used when the Caliper tool is run.

caliperSampling The sampling rate in samples/pixel for x and y, to be used on the calipers. For example, the following calculates the number of caliper samples in the x direction:

If xSize is **caliperSize.x()** in image coordinates, then the effective number of samples in the x direction will be -

c_Int16(xSize * **caliperSampling.x()** + 0.5)

angleRange The beginning angle and the ending angle of the angle range where calipers will be placed along the expected circle perimeter. Angles are measured from the x-axis in the positive direction. The default is the full angular range.

interpolationMethod
The interpolation method used with the calipers. Must be one of the **ccAffineSamplingParams::Interpolation** enums. The default is *eBilinear*.

See the **ccAffineSamplingParams** reference page.

circleFitParams The run-time parameters to use with the Fitting tool.

centrifugal When set true, the caliper search direction is outward from the circle center. When set false, the caliper search direction is inward. The default is true.

startPose A transform that maps the expected circle from circle space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *ccCircleFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

placeCalipersSymmetrically
True specifies that the calipers should be placed at the center of each evenly spaced section of the *angleRange*. False specifies that the calipers be placed at the start of each evenly spaced section. The default is true.

Note: The default mode (true) is always the appropriate choice for placing the calipers. This is flag only exists for backward compatibility. Users who do not require backwards compatibility should not change the state of this flag.

Throws
ccCaliperFinderBaseDefs::BadParams
If *numCalipers* is not zero and is less than **minNumCalipers()**,
or if *caliperSize* is less than (0, 0),
or if *caliperSampling* is less than or equal to (0, 0),
or if *angleRange* is empty,
or if *startPose* is singular.

- `virtual ~ccCaliperCircleFinderAutoRunParams() {}`

Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperCircleFinderAutoRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperCircleFinderAutoRunParams** object to compare to this object.

Public Member Functions

expectedCircle

```
const ccCircle &expectedCircle() const;

void expectedCircle(const ccCircle &expectedCircle);
```

The circle you expect to find in images processed by the Circle Finder tool. The expected circle should be close to actual circles in the images.

- `const ccCircle &expectedCircle() const;`

  Returns the expected circle.

- `void expectedCircle(const ccCircle &expectedCircle);`

  Sets the expected circle.

  Parameters

  expectedCircle The new expected circle.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If *expectedCircle* radius is 0.

angleRange

```
const ccAngleRange &angleRange() const;

void angleRange(const ccAngleRange &angleRange);
```

The beginning angle and the ending angle of the angle range where calipers will be placed along the expected circle perimeter. Angles are measured from the x-axis in the positive direction. The default is the full angular range.

- `const ccAngleRange &angleRange() const;`

  Returns the angle range.

- `void angleRange(const ccAngleRange &angleRange);`

  Sets a new angle range.

  Parameters
  angleRange The new angle range.

  Throws
  ccCaliperFinderBaseDefs::BadParams
  If *angleRange* is empty.

circleFitParams

```
const ccCircleFitParams &circleFitParams() const;

void circleFitParams(
  const ccCircleFitParams &circleFitParams);
```

- `const ccCircleFitParams &circleFitParams() const;`

  Returns the circle fit parameters.

- ```
  void circleFitParams(
    const ccCircleFitParams &circleFitParams);
  ```

  Sets new circle fit parameters.

  Parameters
  circleFitParams The new circle fit parameters.
  ```

ccCaliperCircleFinderAutoRunParams

centrifugal

```
bool centrifugal() const;
void centrifugal(bool centrifugal);
```

When set true, the caliper search direction is outward from the circle center. When set false, the caliper search direction is inward. The default is true.

- `bool centrifugal() const;`

  Returns the centrifugal setting, true or false.

- `void centrifugal(bool centrifugal);`

  Sets the caliper search direction.

  Parameters

  centrifugal The new caliper search direction.

minNumCalipers

```
virtual c_Int16 minNumCalipers() const;
```

Returns the minimum number of calipers required (3).

placeCalipersSymmetrically

```
bool placeCalipersSymmetrically() const;
void placeCalipersSymmetrically(bool symmetrically);
```

True specifies that the calipers should be placed at the center of each evenly spaced section of the *angleRange*. False specifies that the calipers be placed at the start of each evenly spaced section. The default is true.

Notes

The default mode (true) is always the appropriate choice for placing the calipers. This is flag only exists for backward compatibility. Users who do not require backwards compatibility should not change the state of this flag.

- `bool placeCalipersSymmetrically() const;`

  Returns the current specification, true or false.

- `void placeCalipersSymmetrically(bool symmetrically);`

  Sets a new specification.

Parameters

symmetrically The new specification.

Protected Member Functions

computeAffineSamplingParams_

```
virtual void computeAffineSamplingParams_();
```

Computes the affine sampling parameters of the calipers to be run from the current auto run parameters.

Throws

ccCaliperFinderBaseDefs::BadParams
If any members have invalid values.

ccCaliperCircleFinderAutoRunParams

ccCaliperCircleFinderManualRunParams

```
#include <ch_cvl/clpfind.h>

class ccCaliperCircleFinderManualRunParams :
                    public ccCaliperFinderBaseManualRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Encapsulates the Circle Finder tool run-time parameters for running the tool in *manual* mode.

Constructors/Destructors

ccCaliperCircleFinderManualRunParams

```
ccCaliperCircleFinderManualRunParams (
  const ccCircle &expectedCircle = ccCircle(),
  const cmStd vector<ccAffineSamplingParams>& affParams =
                  cmStd vector<ccAffineSamplingParams>(),
  const cmStd vector<ccCaliperRunParams>& clpParams =
                      cmStd vector<ccCaliperRunParams>(),
  const ccCircleFitParams &circleFitParams =
                                        ccCircleFitParams(),
  const cc2Xform &startPose = cc2Xform(),
  bool decrementNumIgnore = true);

virtual ~ccCaliperCircleFinderManualRunParams() {}
```

- ```
  ccCaliperCircleFinderManualRunParams (
    const ccCircle &expectedCircle = ccCircle(),
    const cmStd vector<ccAffineSamplingParams>& affParams =
                    cmStd vector<ccAffineSamplingParams>(),
    const cmStd vector<ccCaliperRunParams>& clpParams =
                        cmStd vector<ccCaliperRunParams>(),
    const ccCircleFitParams &circleFitParams =
                                          ccCircleFitParams(),
    const cc2Xform &startPose = cc2Xform(),
    bool decrementNumIgnore = true);
  ```

  Constructs a circle finder manual run-time parameters object.
  ```

Notes

Default constructed objects do not contain valid run-time parameters.

Parameters

expectedCircle The circle you expect the tool to find in images.

affParams A vector of affine rectangle sampling parameters, one parameter set for each caliper.

clpParams A vector of caliper run-time parameters, one parameter set for each caliper.

circleFitParams The run-time parameters to use with the Fitting tool.

startPose A transform that maps the expected circle from circle space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *ccCircleFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

Throws

ccCaliperFinderBaseDefs::BadParams
If **affParams.size()** is not equal to **clpParams.size()**, or if params sizes are not zero and less than **minNumCalipers()**, or if *startPose* is singular.

- `virtual ~ccCaliperCircleFinderManualRunParams() {}`

Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperCircleFinderManualRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperCircleFinderManualRunParams** object to compare to this object.

Public Member Functions

expectedCircle

```
const ccCircle &expectedCircle() const;

void expectedCircle(const ccCircle &expectedCircle);
```

The circle you expect to find in images processed by the Circle Finder tool. The expected circle should be close to actual circles in the images.

- `const ccCircle &expectedCircle() const;`

  Returns the expected circle.

- `void expectedCircle(const ccCircle &expectedCircle);`

  Sets the expected circle.

  Parameters

  expectedCircle The new expected circle.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If *expectedCircle* radius is 0.

circleFitParams

```
const ccCircleFitParams &circleFitParams() const;

void circleFitParams(
  const ccCircleFitParams &circleFitParams);
```

- `const ccCircleFitParams &circleFitParams() const;`

  Returns the circle fitter run-time parameters.

- ```
  void circleFitParams(
    const ccCircleFitParams &circleFitParams);
  ```

  Sets new circle fit run-time parameters.

  Parameters

  circleFitParams The new circle fit run-time parameters.
  ```

ccCaliperCircleFinderManualRunParams

minNumCalipers `virtual c_Int16 minNumCalipers() const;`

Returns the minimum number of calipers required (3).

ccCaliperCircleFinderResult

```
#include <ch_cvl/clpfind.h>

class ccCaliperCircleFinderResult :
                                public ccCaliperFinderBaseResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the results from the Circle Finder tool.

Constructors/Destructors

ccCaliperCircleFinderResult

```
ccCaliperCircleFinderResult() {}

virtual ~ccCaliperCircleFinderResult() {}
```

- `ccCaliperCircleFinderResult() {}`

  Default constructor. Creates a default constructed (unfound) circle result object.

- `virtual ~ccCaliperCircleFinderResult() {}`

  Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseResult& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperCircleFinderResult** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperCircleFinderResult** object to compare to this object.

Public Member Functions

found

```
virtual bool found() const;
```

Returns true if a circle was found. Returns false otherwise.

A circle is found if its computed error is less than the error *threshold* specified in the Circle Fitting tool parameters. See **ccCircleFitParams::threshold()**.

circleFit

```
const ccCircleFitResults &circleFit() const;
```

Returns the circle fitting result.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

circle

```
const ccCircle &circle() const;
```

Returns the computed circle.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

position

```
const cc2Vect &position() const;
```

Returns the position of the computed circle.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

radius

```
double radius() const;
```

Returns the radius of the computed circle.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

circleFitTime

```
double circleFitTime() const;
```

Returns the circle fitting time, in seconds.

ccCaliperCircleFinderResult

ccCaliperCorrelationResultSet

```
#include <ch_cvl/caliper.h>

class ccCaliperCorrelationResultSet :
                                    public ccCaliperBaseResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the results returned by a single application of the Caliper tool to an input image in correlation mode. Results common to both modes can be found in **ccCaliperBaseResultSet**.

Constructors/Destructors

ccCaliperCorrelationResultSet

```
ccCaliperCorrelationResultSet ();
```

Constructs a **ccCaliperCorrelationResultSet** with no results.

operator==

```
bool operator== (const ccCaliperCorrelationResultSet& that)
  const;
```

Two **ccCaliperCorrelationResultSet** objects are equal if and only if the following items are all the same: the vectors of **ccResult**s; the caliper data; the user-specified origin in client coordinates; the unit vectors in the search direction; the run mode; the result angles; and the clipping status.

Public Member Functions

peakDetectTime

```
double peakDetectTime () const;
```

Returns the amount of time, in seconds, required to perform peak detection.

ccCaliperCorrelationResultSet

runMode

```
ccCaliperDefs::RunMode runMode();
```

Returns the mode in which the caliper was run to generate this **ccCaliperCorrelationResultSet**. The returned value is one of the following values:

ccCaliperDefs::eEdge
ccCaliperDefs::eNormalizedCorrelation
ccCaliperDefs::eRelativeCorrelation

draw

```
void draw(
  ccGraphicList& graphList,
  c_UInt32 drawMode = ccCaliperDefs::eDrawStandard,
  double plotHeight = ccCaliperDefs::kDefaultPlotHeight)
                                                    const;
```

Appends result graphics for all the results in this **ccCaliperCorrelationResultSet** to the supplied **ccGraphicList**.

Parameters

graphList — The graphics list to append the graphics to.

drawMode — The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccCaliperDefs::eDrawEdgeLine
ccCaliperDefs::eDrawLabel
ccCaliperDefs::eDrawProjFilter
ccCaliperDefs::eDrawProjectionRegion
ccCaliperDefs::eDrawStandard
ccCaliperDefs::eDrawRawEdges

plotHeight — The height of the projection and filter plots, in client coordinates.

ccCaliperCorrelationRunParams

```
#include <ch_cvl/caliper.h>

class ccCaliperCorrelationRunParams:
  public ccCaliperBaseRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the run-time parameters for the Caliper tool used in correlation mode, including the reference image.

Constructors/Destructors

ccCaliperCorrelationRunParams

```
ccCaliperCorrelationRunParams();

ccCaliperCorrelationRunParams(
  const ccPelBuffer_const<c_UInt8>& refSrc,
  ccCaliperDefs::RunMode mode =
  ccCaliperDefs::eNormalizedCorrelation,
  const cc2Vect& origin = cc2Vect(0,0));

ccCaliperCorrelationRunParams(
  const ccPelBuffer_const<c_UInt8>& src,
  ccAffineSamplingParams affSamp,
  ccCaliperDefs::RunMode mode =
  ccCaliperDefs::eNormalizedCorrelation,
  const cc2Vect& origin = cc2Vect(0,0));

ccCaliperCorrelationRunParams(
  const ccPelBuffer_const<c_UInt32>& refSrc,
  int nPelsPerBin,
```

```
ccCaliperDefs::RunMode mode =
ccCaliperDefs::eNormalizedCorrelation,
const cc2Vect& origin = cc2Vect(0,0));
```

- `ccCaliperCorrelationRunParams();`

  Constructs a **ccCaliperCorrelationRunParams** with no reference image. This **ccCaliperCorrelationRunParams** will not be usable until you supply a reference image.

- ```
  ccCaliperCorrelationRunParams(
    const ccPelBuffer_const<c_UInt8>& refSrc,
    ccCaliperDefs::RunMode mode =
    ccCaliperDefs::eNormalizedCorrelation,
    const cc2Vect& origin = cc2Vect(0,0));
  ```

  Constructs a **ccCaliperCorrelationRunParams** using the supplied one-dimensional image as the reference image. You must specify the location of the origin in client coordinates.

  Parameters

  refSrc — The image to use. *refSrc* must have an y-dimension equal to 1 pixel.

  mode — The correlation mode to use. *mode* must be one of the following values:

  ccCaliperDefs::eNormalizedCorrelation
  ccCaliperDefs::eRelativeCorrelation

  origin — The origin of the reference image. Correlation peaks will be reported with respect to this origin. *origin* must be specified in *refSrc*'s client coordinate system. The y-coordinate of *origin* is ignored.

  Notes

  The default value of the peak separation threshold is set to the width of the *refSrc*.

  Throws

  ccCaliperDefs::BadParams
  mode is not *ccCaliperDefs::eNormalizedCorrelation* or *ccCaliperDefs::eRelativeCorrelation* or *refSrc.height()* is not 1.

- ```
  ccCaliperCorrelationRunParams(
    const ccPelBuffer_const<c_UInt8>& src,
    ccAffineSamplingParams affSamp,
  ```

```
ccCaliperDefs::RunMode mode =
ccCaliperDefs::eNormalizedCorrelation,
const cc2Vect& origin = cc2Vect(0,0));
```

Constructs a **ccCaliperCorrelationRunParams**. The reference image is constructed by applying the supplied affine sampling parameters to the supplied two-dimensional image. You must specify the location of the origin in client coordinates.

Parameters

src — The image to use to generate the reference image.

affSamp — The affine sampling parameters with which to sample *src*.

mode — The correlation mode to use. *mode* must be one of the following values:

ccCaliperDefs::eNormalizedCorrelation
ccCaliperDefs::eRelativeCorrelation

origin — The origin of the reference image. Correlation peaks will be reported with respect to this origin. *origin* must be specified in *src's* client coordinate system. The origin is transformed to the projection image produced by applying *affSamp* to src.

Notes

The default value of the peak separation threshold is set to the width of the image produced by applying *affSamp* to *src*.

Throws

ccCaliperDefs::BadParams

mode is not *ccCaliperDefs::eNormalizedCorrelation* or *ccCaliperDefs::eRelativeCorrelation*.

- ```
  ccCaliperCorrelationRunParams(
    const ccPelBuffer_const<c_UInt32>& refSrc,
    int nPelsPerBin,
    ccCaliperDefs::RunMode mode =
    ccCaliperDefs::eNormalizedCorrelation,
    const cc2Vect& origin = cc2Vect(0,0));
  ```

  Constructs a **ccCaliperCorrelationRunParams** using the supplied one-dimensional image as the reference image after normalizing each pixel of the image using the supplied normalization value. You must specify the location of the origin in client coordinates.
  ```

Parameters

refSrc — The image to use. *refSrc* must have a y-dimension equal to 1 pixel.

nPelsPerBin — The number of pixels that were summed to produce each value in *refSrc*. Each pixel value in *refSrc* is divided by *nPelsPerBin*.

mode — The correlation mode to use. *mode* must be one of the following values:

ccCaliperDefs::eNormalizedCorrelation
ccCaliperDefs::eRelativeCorrelation

origin — The origin of the reference image. Correlation peaks will be reported with respect to this origin. *origin* must be specified in *refSrc*'s client coordinate system. The y-coordinate of *origin* is ignored.

Throws

ccCaliperDefs::BadParams
mode is not *ccCaliperDefs::eNormalizedCorrelation* or *ccCaliperDefs::eRelativeCorrelation*; *refSrc.height()* is not 1; or *nPelsPerBin* is less than 1.

Public Member Functions

ref1DCorrelation

```
void ref1DCorrelation(
  const ccPelBuffer_const<c_UInt8>& ref1D,
  const cc2Vect& origin= cc2Vect(0,0));

void ref1DCorrelation(cmStd vector<ccIPair> refValues,
  int smooth=0, double origin = 0.0);

void ref1DCorrelation(cmStd vector<c_UInt8> refValues,
  double origin= 0.0);

void ref1DCorrelation(
  ccCaliperDefs::PreDefined1DSignal refSignal,
  c_UInt32 width, c_UInt8 max, c_UInt8 min, int smooth=0,
  double origin = 0.0);
```

-
```
void ref1DCorrelation(
  const ccPelBuffer_const<c_UInt8>& ref1D,
  const cc2Vect& origin= cc2Vect(0,0));
```

Sets this **ccCaliperCorrelationRunParams**'s reference image to be the supplied one-dimensional image. You must specify the location of the origin in client coordinates.

Parameters

ref1D — The image to use. *ref1D* must have a y-dimension equal to 1 pixel.

origin — The origin of the reference image. Correlation peaks will be reported with respect to this origin. *origin* must be specified in *refSrc*'s client coordinate system. The y-coordinate of *origin* is ignored.

Notes

The default value of the peak separation threshold is set to the width of the *refSrc*.

Throws

ccFiltErr::BadImageSize
ref1D.height() is not 1 or *ref1D*.width() is less than 1.

- `void ref1DCorrelation(cmStd vector<ccIPair> refValues, int smooth=0, double origin = 0.0);`

Sets this **ccCaliperCorrelationRunParams**'s reference image to be the supplied vector of run-length codes. Any specified smoothing value is used as input to the Gaussian Sampling tool to smooth the image produced from the list of run-length codes. The origin is specified in the image coordinate system of the image produced from the list of run-length codes.

Parameters

refValues — A vector of run-length codes. Each element of *refValues* is interpreted as follows: *refValues.x()* is the number of pixels in the run and *refValues.y()* is the pixel value of the run.

smooth — The smoothing value.

origin — The reference image origin, specified in image coordinates of the image produced from *refValues*.

Notes

The default value of the peak separation threshold is set to the width of the *refSrc*.

Throws

ccFiltErr::BadImageSize
refValues contains less than 1 element.

ccCaliperDefs::BadParams
smooth is less than 0.

- ```
  void ref1DCorrelation(cmStd vector<c_UInt8> refValues,
    double origin= 0.0);
  ```

  Sets this **ccCaliperCorrelationRunParams**'s reference image to be the supplied vector of values. The origin is specified in the image coordinate system of the image produced from the list of values.

  Parameters

  refValues — A vector of 8-bit pixel values.

  origin — The reference image origin, specified in image coordinates of the image produced from *refValues*.

  Notes

  The default value of the peak separation threshold is set to the width of the *refSrc*.

  Throws

  ccFiltErr::BadImageSize
  refValues contains less than 1 element.

- ```
  void ref1DCorrelation(
    ccCaliperDefs::PreDefined1DSignal refSignal,
    c_UInt32 width, c_UInt8 max, c_UInt8 min, int smooth=0,
    double origin = 0.0);
  ```

  Sets this **ccCaliperCorrelationRunParams**'s reference image to be the specified pre-defined reference image. The origin is specified in the image coordinate system of the image.

  If you specify *ccCaliperDefs::ePosGauss* or *ccCaliperDefs::eNegGauss*, this function applies a Gaussian kernel with a sigma equal to one fourth of the width of the image. You can override this kernel size.

  Parameters

  refSignal — The pre-defined reference image. *refSignal* must be one of the following values:

  ccCaliperDefs::ePosPulse
  ccCaliperDefs::eNegPulse
  ccCaliperDefs::ePosGauss
  ccCaliperDefs::eNegGauss

  width — The width of the reference signal. The overall width of the resulting image is equal to *width*+2.

  max — The maximum value of the image. For *ccCaliperDefs::ePosPulse* and *ccCaliperDefs::ePosGauss*, *max* is the value of the peak.

max The minimum value of the image. For *ccCaliperDefs::eNegPulse* and *ccCaliperDefs::eNegGauss*, *max* is the value of the peak.

smooth The smoothing value to apply to produce a Gaussian reference image. Set *smooth* to a positive value to override the default smoothing value. The resulting sigma will be approximately 1.5 times *smooth*.

Throws

ccFiltErr::BadImageSize
width is less than 1.

ccCaliperDefs::BadParams
max is less than *min* or *smooth* is less than 0.

ccFiltErr::BadImageSize
width is less than 1.

acceptThreshold

```
double acceptThreshold() const;

void acceptThreshold(double thresh);
```

- `double acceptThreshold() const;`

  Returns the accept threshold for this **ccCaliperCorrelationRunParams**. Only correlation peaks with correlation scores greater than this threshold are scored.

- `void acceptThreshold(double thresh);`

  Sets the accept threshold for this **ccCaliperCorrelationRunParams**. Only correlation peaks with correlation scores greater than this threshold are scored.

  The default accept threshold is 0.

  Parameters

  thresh The accept threshold. *thresh* must be in the range 0 to 1000.

  Throws

  ccCaliperDefs::BadParams
  thresh is less than 0 or greater than 1000.

Notes

Keep in mind that correlation scores differ depending on the type of correlation. For normalized correlation, scores range between 0 and 1000. For relative correlation, scores range between 0 and the maximum value that can be stored by a pixel in the input image (255 for 8-bit images).

runMode

```
ccCaliperDefs::RunMode runMode() const;
void runMode(ccCaliperDefs::RunMode mode, double thresh);
```

- `ccCaliperDefs::RunMode runMode() const;`

  Returns the type of correlation for this **ccCaliperCorrelationRunParams**.

- `void runMode(ccCaliperDefs::RunMode mode, double thresh);`

  Sets the type of correlation for this **ccCaliperCorrelationRunParams**.

  Parameters

  mode — The correlation mode. *mode* must be one of the following values:

  ccCaliperDefs::eNormalizedCorrelation
  ccCaliperDefs::eRelativeCorrelation

  thresh — The accept threshold for this **ccCaliperCorrelationRunParams**. See **acceptThreshold()** on page 799 for information on the accept threshold.

  Throws

  ccCaliperDefs::BadParams
  mode is not *ccCaliperDefs::eNormalizedCorrelation* or *ccCaliperDefs::eRelativeCorrelation*.

maxNumResults

```
c_Int16 maxNumResults () const;
void maxNumResults (c_Int16 num);
```

- `c_Int16 maxNumResults () const;`

  Returns the maximum number of results that this **ccCaliperCorrelationRunParams** is configured to return.

  `void maxNumResults (c_Int16 num);`

  Sets the maximum number of results that this **ccCaliperCorrelationRunParams** is configured to return.

Parameters

num The maximum number of results to return. If you specify 0 for *num*, the tool will perform the projection, filtering, and peak detection steps, but it will not produce any scored results.

Throws

ccCaliperDefs::BadParams
num is less than 0.

scoringMethods

```
const cmStd vector<ccCaliperScore*> scoringMethods() const;

void scoringMethods (
  const cmStd vector<ccCaliperScore*>& methods);
```

- `const cmStd vector<ccCaliperScore*> scoringMethods() const;`

  Returns a vector of **ccCaliperScore** objects that define how correlation peaks are scored.

- ```
  void scoringMethods (
    const cmStd vector<ccCaliperScore*>& methods);
  ```

  Sets the list of **ccCaliperScore** objects that determine how correlation peaks are scored. Any existing list of **ccCaliperScore** objects is discarded.

  The only legal type of scoring object for correlation mode is **ccScoreContrast**.

  Parameters

  methods The scoring methods to set.

  Notes

  If *methods* contains no objects (has a length of 0), then any attempt to run the Caliper tool with a value for the maximum number of results other than 0 will throw an error.

  A default-constructed **ccCaliperCorrelationRunParams** object contains a single **ccScoreContrast** scoring method.

  Throws

  ccCaliperDefs::BadParams
  An element of *method* is not of type **ccScoreContrast**.
  ```

peakSeparationThreshold

```
double peakSeparationThreshold() const;

void peakSeparationThreshold(double peakSep);
```

- `double peakSeparationThreshold() const;`

  Returns the peak separation threshold configured for this **ccCaliperCorrelationRunParams**.

- `void peakSeparationThreshold(double peakSep);`

  Sets the peak separation threshold configured for this **ccCaliperCorrelationRunParams**.

  The peak separation threshold is the minimum distance, in client coordinate system units, between separate peaks. Peaks that are closer together than the threshold are treated as a single peak; the peak with the lower score is discarded.

  Parameters

  peakSep The peak separation threshold.

ccCaliperDefs

```
#include <ch_cvl/caliper.h>

class ccCaliperDefs;
```

A name space class for the Caliper tool.

Enumerations

Interpolation

```
enum Interpolation;
```

This enumeration defines the sampling types supported by the Caliper tool.

Value	Meaning
eNone	Nearest neighbor sampling is used to construct the projection image.
eBilinear	Bilinear interpolation is used to construct the projection image.
kDefaultInterpolation	The default sampling method, as defined in *caliper.h.*

RunMode

```
enum RunMode;
```

This enumeration defines the modes supported by the Caliper tool.

Value	Meaning
eEdge	Features are located by detecting and scoring of edge peaks and edge pair peaks.
eNormalizedCorrelation	Features are located be detecting and scoring normalized correlation peaks between run-time and reference images.
eRelativeCorrelation	Features are located be detecting and scoring relative correlation peaks between run-time and reference images.
kDefaultRunMode	The default mode, as defined in *caliper.h.*

PreDefined1DSignal

```
enum PreDefined1DSignal;
```

This enumeration defines the pre-defined reference signals available for correlation mode.

Value	Meaning
ePosPulse	A boxcar signal going from low to high to low.
eNegPulse	A boxcar signal going from high to low to high.
ePosGauss	A boxcar signal going from low to high to low which has been smoothed using a Gaussian filter.
eNegGauss	A boxcar signal going from high to low to high which has been smoothed using a Gaussian filter.

DrawMode

```
enum DrawMode;
```

This enumeration defines the types of result graphics you can draw for Caliper tool results.

Value	Meaning
eDrawEdgeLine	Draws a line at the location of each detected edge.
eDrawLabel	Draws a text label at each detected edge.
eDrawProjFilter	Draws a graphical representation of the filtered projection image and projection image at the top and bottom (respectively) of the bounding box for the result.
eDrawProjectionRegion	Draws a box describing the projection region.
eDrawStandard	Draws a labeled line for each detected edge, and draws a box around the projection region.

Notes

Edges and labels are drawn in **ccColor::greenColor()**, projected and filtered images are drawn in **ccColor::cyanColor()**, bounding boxes are drawn in **ccColor::magentaColor()**, and projection region is drawn in **ccColor::blueColor()**.

ResultSetDrawMode

```
enum ResultSetDrawMode;
```

This enumeration defines the types of result graphics you can draw for Caliper tool results.

Value	Meaning
eDrawRawEdges	Draws a **ccColor::redColor()** line at each edge detected in the projection image.

```
enum { kDefaultPlotHeight = 30 };
```

ccCaliperDefs

ccCaliperDesiredEdge

```
#include <ch_cvl/caliper.h>

class ccCaliperDesiredEdge;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class describes a single edge. This class is constructed automatically by the Caliper tool. You specify the location and polarity of the model edge or edges when you construct a **ccCaliperRunParams**.

Note You should not construct a **ccCaliperDesiredEdge** directly.

Constructors/Destructors

ccCaliperDesiredEdge

```
ccCaliperDesiredEdge (double position = 0.0,
  ceEdgePolarity polarity = ceDontCare);
```

Construct this edge using the specified parameters.

Parameters

position The expected position of this edge with respect to the edge model origin in input image client coordinate system units.

polarity The expected polarity of the edge. *polarity* must be one of the following values:

ceDarkToLight
ceLightToDark
ceDontCare

A value of *ceDontCare* means the edge has either polarity.

Public Member Functions

position

```
double position () const;
void position (double pos);
```

- `double position () const;`

  Return the position of this edge with respect to the model origin. The position is a signed value in client coordinate system units.

- `void position (double pos);`

  Sets the position of this edge with respect to the model origin. The position is a signed value in client coordinate units.

  Parameters

  pos The position to set.

  Notes

  The positions returned by the Caliper tool's **cfCaliperRun()** function indicate the distance between the model origin and the center of the caliper window (or another, user-supplied origin)

polarity

```
ceEdgePolarity polarity () const;
void polarity (ceEdgePolarity pol);
```

- `ceEdgePolarity polarity () const;`

  Returns the polarity of this edge. The polarity is defined as being measured in the ascending x-axis direction of the projection image. The value returned by this function is one of the following:

  ceDarkToLight
  ceLightToDark
  ceDontCare

- `void polarity (ceEdgePolarity pol);`

  Sets the polarity of this edge. The polarity is defined as being measured in the ascending x-axis direction of the projection image.

  Parameters

  pol The polarity to set. *pol* must be one of the following values:

ceDarkToLight
ceLightToDark
ceDontCare

ccCaliperDesiredEdge

ccCaliperEllipseFinderAutoRunParams

```
#include <ch_cvl/clpfind.h>

class ccCaliperEllipseFinderAutoRunParams :
                        public ccCaliperFinderBaseAutoRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Encapsulates the Ellipse Finder tool run-time parameters for running the tool in *auto* mode.

Constructors/Destructors

ccCaliperEllipseFinderAutoRunParams

```
ccCaliperEllipseFinderAutoRunParams();

ccCaliperEllipseFinderAutoRunParams (
  const ccEllipse2 &expectedEllipse,
  c_Int16 numCalipers = 0,
  const ccDPair &caliperSize = ccDPair(0, 0),
  const ccCaliperRunParams &caliperRunParams =
                                      ccCaliperRunParams(),
  const ccDPair &caliperSampling = ccDPair(1, 1),
  const ccAngleRange &angleRange =
                            ccAngleRange::FullAngleRange(),
  ccAffineSamplingParams::Interpolation
                         interpolationMethod =
                         ccAffineSamplingParams::eBilinear,
  const ccEllipseFitParams &ellipseFitParams =
                                      ccEllipseFitParams(),
  bool centrifugal = true,
  const cc2Xform& startPose = cc2Xform(),
  bool decrementNumIgnore = true);

virtual ~ccCaliperEllipseFinderAutoRunParams() {}
```

- `ccCaliperEllipseFinderAutoRunParams();`

  Default constructor.

Notes

A default constructed object is not valid for running.

- ```
  ccCaliperEllipseFinderAutoRunParams (
    const ccEllipse2 &expectedEllipse,
    c_Int16 numCalipers = 0,
    const ccDPair &caliperSize = ccDPair(0, 0),
    const ccCaliperRunParams &caliperRunParams =
                                          ccCaliperRunParams(),
    const ccDPair &caliperSampling = ccDPair(1, 1),
    const ccAngleRange &angleRange =
                              ccAngleRange::FullAngleRange(),
    ccAffineSamplingParams::Interpolation
                           interpolationMethod =
                           ccAffineSamplingParams::eBilinear,
    const ccEllipseFitParams &ellipseFitParams =
                                          ccEllipseFitParams(),
    bool centrifugal = true,
    const cc2Xform& startPose = cc2Xform(),
    bool decrementNumIgnore = true);
  ```

Constructs a ellipse finder run-time parameters object.

Parameters

expectedEllipse The ellipse you expect the tool to find in images.

numCalipers The number of calipers the tool should use to find the ellipse.

caliperSize The caliper size, width and height in client coordinates. All calipers are the same size.

caliperRunParams
The run-time parameters object to be used when the Caliper tool is run.

caliperSampling The sampling rate in samples/pixel for x and y, to be used on the calipers. For example, the following calculates the number of caliper samples in the x direction:

If xSize is **caliperSize.x()** in image coordinates, then the effective number of samples in the x direction will be -

c_Int16(xSize * **caliperSampling.x()** + 0.5)

angleRange The beginning angle and the ending angle of the angle range where calipers will be placed along the expected ellipse perimeter. Angles are measured from the x-axis in the positive direction around a unit circle centered inside the ellipse. The
  ```

angle range along the unit circle perimeter is projected outward onto the ellipse perimeter. See the *Shape Finder Tool* chapter of the *Vision Tools Guide* for details.

The default is the full angular range.

interpolationMethod
The interpolation method used with the calipers. Must be one of the **ccAffineSamplingParams::Interpolation** enums. The default is *eBilinear*.

See the **ccAffineSamplingParams** reference page.

ellipseFitParams The run-time parameters to use with the Fitting tool.

centrifugal When set true, the caliper search direction is outward from the ellipse center. When set false, the caliper search direction is inward. The default is true.

startPose A transform that maps the expected ellipse from ellipse space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *ccEllipseFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

Throws

ccCaliperFinderBaseDefs::BadParams
If *numCalipers* is not zero and is less than **minNumCalipers()**,
or if *caliperSize* is less than (0, 0),
or if *caliperSampling* is less than or equal to (0, 0),
or if *angleRange* is empty,
or if *startPose* is singular.

- `virtual ~ccCaliperEllipseFinderAutoRunParams() {}`

Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperEllipseFinderAutoRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperEllipseFinderAutoRunParams** object to compare to this object.

Public Member Functions

expectedEllipse

```
ccEllipse2 &expectedEllipse2() const;

void expectedEllipse2(const ccEllipse2 &expectedEllipse);
```

The ellipse you expect to find in images processed by the Ellipse Finder tool. The expected ellipse should be close to actual ellipses in the images.

- `ccEllipse2 &expectedEllipse2() const;`

  Returns the expected ellipse.

- `void expectedEllipse2(const ccEllipse2 &expectedEllipse);`

  Sets a new expected ellipse.

  Parameters

  expectedEllipse The new expected ellipse.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If an *expectedEllipse* radius is 0.

angleRange

```
const ccAngleRange &angleRange() const;

void angleRange(const ccAngleRange &angleRange);
```

The beginning angle and the ending angle of the angle range where calipers will be placed along the expected ellipse perimeter. Angles are measured from the x-axis in the positive direction around a unit circle centered inside the ellipse. The angle range along the unit circle perimeter is projected outward onto the ellipse perimeter. See the *Shape Finder Tool* chapter of the *Vision Tools Guide* for details.

The default is the full angular range.

Notes

This angle range is not in client coordinates. It is in the intrinsic coordinates of the ellipse to be consistent with the **ccEllipseArc** and **ccEllipseAnnulusSection** classes. Therefore, be careful when the ellipse's aspect ratio changes because of a transform since the angle range will mean something very different. For more information, see the **ccEllipse**, **ccEllipseArc**, and **ccEllipseAnnulusSection** reference pages.

- `const ccAngleRange &angleRange() const;`

  Returns the angle range.

- `void angleRange(const ccAngleRange &angleRange);`

  Sets a new angle range.

  Parameters
  angleRange The new angle range.

  Throws
  ccCaliperFinderBaseDefs::BadParams
  If *angleRange* is empty.

ellipseFitParams

```
const ccEllipseFitParams &ellipseFitParams() const;

void ellipseFitParams(
  const ccEllipseFitParams &ellipseFitParams);
```

- `const ccEllipseFitParams &ellipseFitParams() const;`

  Returns the ellipse fit parameters.

- `void ellipseFitParams(`
  `const ccEllipseFitParams &ellipseFitParams);`

  Sets new ellipse fit parameters.

  Parameters
  ellipseFitParams The new ellipse fit parameters.

centrifugal

```
bool centrifugal() const;
void centrifugal(bool centrifugal);
```

When set true, the caliper search direction is outward from the ellipse center. When set false, the caliper search direction is inward. The default is true.

- `bool centrifugal() const;`

  Returns the centrifugal setting, true or false.

- `void centrifugal(bool centrifugal);`

  Sets the caliper search direction.

  Parameters
  centrifugal The new caliper search direction.

minNumCalipers

```
virtual c_Int16 minNumCalipers() const;
```

Returns the minimum number of calipers required (5).

Deprecated Members

The following have been deprecated and are provided here for backward compatibility only.

expectedEllipse

```
const ccEllipse &expectedEllipse() const;
void expectedEllipse(const ccEllipse &expectedEllipse);
```

The ellipse you expect to find in images processed by the Ellipse Finder tool. The expected ellipse should be close to actual ellipses in the images.

ccCaliperEllipseFinderAutoRunParams

```
ccCaliperEllipseFinderAutoRunParams (
  const ccEllipse &expectedEllipse,
  c_Int16 numCalipers = 0,
  const ccDPair &caliperSize = ccDPair(0, 0),
  const ccCaliperRunParams &caliperRunParams =
                                        ccCaliperRunParams(),
  const ccDPair &caliperSampling = ccDPair(1, 1),
  const ccAngleRange &angleRange =
                              ccAngleRange::FullAngleRange(),
  ccAffineSamplingParams::Interpolation
                           interpolationMethod =
                           ccAffineSamplingParams::eBilinear,
  const ccEllipseFitParams &ellipseFitParams =
                                        ccEllipseFitParams(),
  bool centrifugal = true,
  const cc2Xform& startPose = cc2Xform(),
  bool decrementNumIgnore = true);
```

Constructor.

ccCaliperEllipseFinderAutoRunParams

ccCaliperEllipseFinderManualRunParams

```
#include <ch_cvl/clpfind.h>

class ccCaliperEllipseFinderManualRunParams :
                      public ccCaliperFinderBaseManualRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Encapsulates the Ellipse Finder tool run-time parameters for running the tool in *manual* mode.

Constructors/Destructors

ccCaliperEllipseFinderManualRunParams

```
ccCaliperEllipseFinderManualRunParams() :
   ccCaliperFinderBaseManualRunParams(cc2Xform(), true) {}

ccCaliperEllipseFinderManualRunParams (
  const ccEllipse2 &expectedEllipse,
  const cmStd vector<ccAffineSamplingParams>& affParams =
                    cmStd vector<ccAffineSamplingParams>(),
  const cmStd vector<ccCaliperRunParams>& clpParams =
                        cmStd vector<ccCaliperRunParams>(),
  const ccEllipseFitParams &ellipseFitParams =
                                       ccEllipseFitParams(),
  const cc2Xform &startPose = cc2Xform(),
  bool decrementNumIgnore = true);

virtual ~ccCaliperEllipseFinderManualRunParams() {}
```

- ```
  ccCaliperEllipseFinderManualRunParams() :
     ccCaliperFinderBaseManualRunParams(cc2Xform(), true) {}
  ```

  Default constructor.

  Notes

  A default constructed object is not valid for running.
  ```

- ```
  ccCaliperEllipseFinderManualRunParams (
    const ccEllipse2 &expectedEllipse,
    const cmStd vector<ccAffineSamplingParams>& affParams =
                        cmStd vector<ccAffineSamplingParams>(),
    const cmStd vector<ccCaliperRunParams>& clpParams =
                           cmStd vector<ccCaliperRunParams>(),
    const ccEllipseFitParams &ellipseFitParams =
                                          ccEllipseFitParams(),
    const cc2Xform &startPose = cc2Xform(),
    bool decrementNumIgnore = true);
  ```

  Constructs a ellipse finder manual run-time parameters object.

  Parameters

  expectedEllipse The ellipse you expect the tool to find in images.

  affParams A vector of affine rectangle sampling parameters, one parameter set for each caliper.

  clpParams A vector of caliper run-time parameters, one parameter set for each caliper.

  ellipseFitParams The run-time parameters to use with the Fitting tool.

  startPose A transform that maps the expected ellipse from ellipse space to its expected pose in the run-time client space.

  decrementNumIgnore
  When true, *ccEllipseFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If **affParams.size()** is not equal to **clpParams.size()**, or if params sizes are not zero and less than **minNumCalipers()**, or if *startPose* is singular.

- ```
  virtual ~ccCaliperEllipseFinderManualRunParams() {}
  ```

  Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperEllipseFinderManualRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperEllipseFinderManualRunParams** object to compare to this object.

Public Member Functions

expectedEllipse2

```
ccEllipse2 expectedEllipse() const;

void expectedEllipse2(const ccEllipse2 &expectedEllipse);
```

The ellipse you expect to find in images processed by the Ellipse Finder tool. The expected ellipse should be close to actual ellipses in the images.

- `ccEllipse2 expectedEllipse() const;`

  Returns the expected ellipse.

- `void expectedEllipse2(const ccEllipse2 &expectedEllipse);}`

  Sets a new expected ellipse.

  Parameters

  expectedEllipse The new expected ellipse.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If an *expectedEllipse* radius is 0.

ellipseFitParams

```
const ccEllipseFitParams &ellipseFitParams() const;

void ellipseFitParams(
  const ccEllipseFitParams &ellipseFitParams);
```

- `const ccEllipseFitParams &ellipseFitParams() const;`

  Returns the ellipse fitter run-time parameters.

- ```
  void ellipseFitParams(
    const ccEllipseFitParams &ellipseFitParams);
  ```

  Sets new ellipse fitter run-time parameters.

  Parameters

  ellipseFitParams The new parameters.

minNumCalipers

```
virtual c_Int16 minNumCalipers() const;
```

Returns the minimum number of calipers required (5).

Deprecated Members

The following have been deprecated and are provided here only for backward compatibility.

expectedEllipse

```
const ccEllipse &expectedEllipse() const;

void expectedEllipse(const ccEllipse &expectedEllipse);
```

The ellipse you expect to find in images processed by the Ellipse Finder tool. The expected ellipse should be close to actual ellipses in the images.

ccCaliperEllipseFinderManualRunParams

```
ccCaliperEllipseFinderManualRunParams (
  const ccEllipse &expectedEllipse,
  const cmStd vector<ccAffineSamplingParams>& affParams =
                  cmStd vector<ccAffineSamplingParams>(),
  const cmStd vector<ccCaliperRunParams>& clpParams =
                        cmStd vector<ccCaliperRunParams>(),
  const ccEllipseFitParams &ellipseFitParams =
                                      ccEllipseFitParams(),
  const cc2Xform &startPose = cc2Xform(),
  bool decrementNumIgnore = true);
```

Constructor.
```

ccCaliperEllipseFinderResult

```
#include <ch_cvl/clpfind.h>

class ccCaliperEllipseFinderResult :
                                    public ccCaliperFinderBaseResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the results from the Ellipse Finder tool.

Constructors/Destructors

ccCaliperEllipseFinderResult

```
ccCaliperEllipseFinderResult() {};

virtual ~ccCaliperEllipseFinderResult() {}
```

- `ccCaliperEllipseFinderResult() {};`

  Default constructor. Creates a default constructed (unfound) ellipse result object.

- `virtual ~ccCaliperEllipseFinderResult() {}`

  Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseResult& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperEllipseFinderResult** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperEllipseFinderResult** object to compare to this object.

Public Member Functions

found

```
virtual bool found() const;
```

Returns true if an ellipse was found. Returns false otherwise.

An ellipse is found if its computed error is less than the error *threshold* specified in the Ellipse Fitting tool parameters. See **ccEllipseFitParams::threshold()**.

ellipseFit

```
const ccEllipseFitResults &ellipseFit() const;
```

Returns the ellipse fitting result.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

ellipse2

```
const ccEllipse2 &ellipse2() const;
```

Returns the found ellipse if **found()** is true.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

position

```
const cc2Vect &position() const;
```

Returns the position of the computed ellipse.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

ellipseFitTime

```
double ellipseFitTime() const;
```

Returns the circle fitting time, in seconds.

Deprecated Members

ellipse

```
const ccEllipse &ellipse() const;
```

Returns the found ellipse if **found()** is true.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

ccCaliperEllipseFinderResult

ccCaliperFinderBaseAutoRunParams

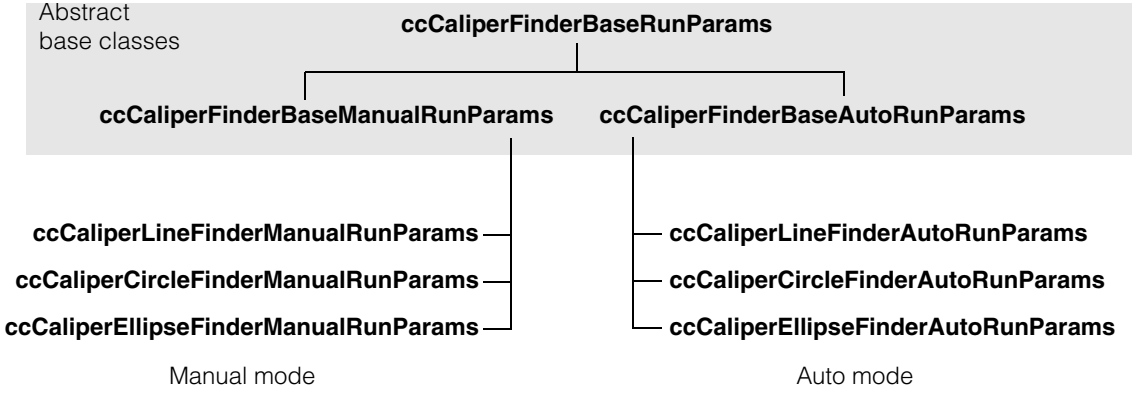
```
#include <ch_cvl/clpfind.h>

class ccCaliperFinderBaseAutoRunParams :
                              public ccCaliperFinderBaseRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This is an abstract base class for the Shape Finder tool run-time parameters. See the following class hierarchy diagram.



Constructors/Destructors

ccCaliperFinderBaseAutoRunParams

```
ccCaliperFinderBaseAutoRunParams (
  c_Int16 numCalipers,
  const ccDPair &caliperSize,
  const ccCaliperRunParams &caliperRunParams,
  const ccDPair &caliperSampling,
  ccAffineSamplingParams::Interpolation
```

```
                                                    interpolationMethod,
  const cc2Xform &startPose,
  bool decrementNumIgnore);

ccCaliperFinderBaseAutoRunParams(
  const ccCaliperFinderBaseAutoRunParams&);

virtual ~ccCaliperFinderBaseAutoRunParams() = 0;
```

-
```
ccCaliperFinderBaseAutoRunParams (
  c_Int16 numCalipers,
  const ccDPair &caliperSize,
  const ccCaliperRunParams &caliperRunParams,
  const ccDPair &caliperSampling,
  ccAffineSamplingParams::Interpolation
                                                    interpolationMethod,
  const cc2Xform &startPose,
  bool decrementNumIgnore);
```

Protected; constructs a finder base auto run-time parameters object.

Parameters

numCalipers The number of calipers the tool should use to find the shape.

caliperSize The caliper size, width and height in client coordinates. All calipers are the same size. Caliper width is also called the search length and caliper height is also called the projection length.

caliperRunParams
The run-time parameters object to be used when the Caliper tool is run.

caliperSampling The sampling rate in samples/pixel for x and y, to be used on the calipers. For example, the following calculates the number of caliper samples in the x direction:

If xSize is **caliperSize.x()** in image coordinates, then the effective number of samples in the x direction will be -

c_Int16(xSize * **caliperSampling.x()** + 0.5)

interpolationMethod
The interpolation method used with the calipers. Must be one of the **ccAffineSamplingParams::Interpolation** enums.

See the **ccAffineSamplingParams** reference page.

startPose A transform that maps the expected shape from shape space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *numIgnore* in the shape fit parameters is decremented for each caliper that fails to find an edge. When false, no decrement occurs. The default is true.

Throws

ccCaliperFinderBaseDefs::BadParams
If *numCalipers* is less than 0,
or if *caliperSize* is less than (0, 0),
or if *caliperSampling* is less than or equal to (0, 0),
or if *startPose* is singular.

- ```
  ccCaliperFinderBaseAutoRunParams(
    const ccCaliperFinderBaseAutoRunParams&);
  ```

  Protected copy constructor.

- ```
  virtual ~ccCaliperFinderBaseAutoRunParams() = 0;
  ```

  Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperFinderBaseRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The object to compare to this object.

operator=

```
ccCaliperFinderBaseAutoRunParams& operator= (
  const ccCaliperFinderBaseAutoRunParams& newObj);
```

Protected assignment operator. Prohibits copying and assignment by base pointers.

Parameters

newObj The new object that is a copy of this object.

Public Member Functions

numCalipers

```
c_Int16 numCalipers() const;
void numCalipers(c_Int16 numCalipers);
```

The number of calipers the tool should use to find the shape. The default is zero which is not valid for running the tool.

- `c_Int16 numCalipers() const;`

  Returns the number of calipers to use.

- `void numCalipers(c_Int16 numCalipers);`

  Sets a new number of calipers.

  Parameters
  numCalipers The new number of calipers.

  Throws
  ccCaliperFinderBaseDefs::BadParams
  If *numCalipers* < **minNumCalipers()**.

caliperWidth

```
double caliperWidth() const;
void caliperWidth(double caliperWidth);
```

The caliper width in pixels.

- `double caliperWidth() const;`

  Returns the caliper width.

- `void caliperWidth(double caliperWidth);`

  Sets a new caliper width.

  Parameters
  caliperWidth The new caliper width.

caliperHeight

```
double caliperHeight() const;
void caliperHeight(double caliperHeight);
```

The caliper height in pixels.

- `double caliperHeight() const;`

  Returns the caliper height.

- `void caliperHeight(double caliperHeight);`

  Sets a new caliper height.

  Parameters
  caliperHeight The new caliper height.

caliperSize

```
const ccDPair &caliperSize() const;
void caliperSize(const ccDPair &caliperSize);
```

The caliper size, width and height in client coordinates. All calipers are the same size. The default is (0, 0) which is not valid for running a tool.

- `const ccDPair &caliperSize() const;`

  Returns the caliper size.

- `void caliperSize(const ccDPair &caliperSize);`

  Sets a new caliper size.

  Parameters
  caliperSize The new caliper size.

  Throws
  ccCaliperFinderBaseDefs::BadParams
  If *caliperSize* is equal to or less than (0., 0.).

caliperRunParams

```
ccCaliperRunParams caliperRunParams() const;

void caliperRunParams(
  const ccCaliperRunParams &caliperRunParams);
```

The run-time parameters object to use when the Caliper tool is run.

- `ccCaliperRunParams caliperRunParams() const;`

  Returns the current run-time parameters.

- `void caliperRunParams(`
  `const ccCaliperRunParams &caliperRunParams);`

  Sets a new run-time parameters object.

  Parameters

  caliperRunParams
  The new run-time parameters.

caliperSampling

```
const ccDPair &caliperSampling() const;

void caliperSampling(const ccDPair &caliperSampling);
```

The sampling rate in samples/pixel for x and y (width and height), to be used on the calipers. See *caliperSampling* in the ctor.

The default is (1, 1).

- `const ccDPair &caliperSampling() const;`

  Returns the caliper sampling values.

- `void caliperSampling(const ccDPair &caliperSampling);`

  Sets new caliper sampling values.

  Parameters

  caliperSampling The new caliper sampling.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If *caliperSampling* is equal to or less than (0., 0.).

interpolationMethod

```
ccAffineSamplingParams::Interpolation
                                    interpolationMethod() const;

void interpolationMethod(
  ccAffineSamplingParams::Interpolation method);
```

The interpolation method used with the calipers. Must be one of the **ccAffineSamplingParams::Interpolation** enums. See the **ccAffineSamplingParams** reference page.

The default is bilinear interpolation.

- ```
  ccAffineSamplingParams::Interpolation
                                    interpolationMethod() const;
  ```

  Returns the current interpolation method.

- ```
  void interpolationMethod(
    ccAffineSamplingParams::Interpolation method);
  ```

  Sets a new interpolation method.

  Parameters

  method The new interpolation method.

affineSamplingParamsList

```
virtual const cmStd
vector<ccAffineSamplingParams> affineSamplingParamsList(
  const cc2XformBase &clientFromImage =
                                       cc2XformLinear()) const;
```

Returns the affine sampling parameters. The parameters are computed if necessary.

Parameters

clientFromImage

The client space to image space transform.

Throws

ccCaliperFinderBaseDefs::BadParams

If any run parameter is invalid,
or if the sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

Notes

These parameters do not reflect the start pose.

Protected Member Functions

dirty

```
void dirty();
```

Sets the dirty flag to true so that affine sampling parameters are recomputed when the tool is run.

computeAffineSamplingParams

```
void computeAffineSamplingParams(
  const cc2XformBase &imageFromClient);
```

This function is for Cognex internal use only.

Throws

ccCaliperFinderBaseDefs::BadParams
If any run parameter is invalid,
or if the sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

computeAffineSamplingParams_

```
virtual void computeAffineSamplingParams_();
```

This function is for Cognex internal use only.

Throws

ccCaliperFinderBaseDefs::BadParams
If any run parameter is invalid,
or if the sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

ccCaliperFinderBaseManualRunParams

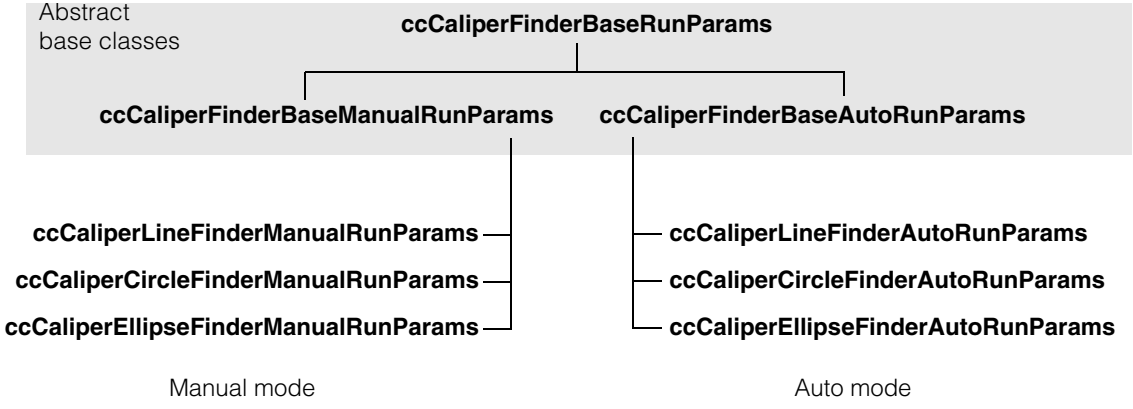
```
#include <ch_cvl/clpfind.h>

class ccCaliperFinderBaseManualRunParams :
                              public ccCaliperFinderBaseRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This is an abstract base class for the Shape Finder tool run-time parameters. See the following class hierarchy diagram.



Constructors/Destructors

ccCaliperFinderBaseManualRunParams

```
ccCaliperFinderBaseManualRunParams (
  const cc2Xform &startPose,
  bool decrementNumIgnore);

ccCaliperFinderBaseManualRunParams (
  const ccCaliperFinderBaseManualRunParams&);

virtual ~ccCaliperFinderBaseManualRunParams() = 0;
```

- ```
  ccCaliperFinderBaseManualRunParams (
    const cc2Xform &startPose,
    bool decrementNumIgnore);
  ```

  Protected; constructs a finder base manual run-time parameters object.

  Parameters

  startPose A transform that maps the expected shape from shape space to its expected pose in the run-time client space. It also maps the caliper affine rectangles into client space.

  decrementNumIgnore
  When true, *numIgnore* in the shape fit parameters is decremented for each caliper that fails to find an edge. When false, no decrement occurs. The default is true.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If *startPose* is singular.

- ```
  ccCaliperFinderBaseManualRunParams (
    const ccCaliperFinderBaseManualRunParams&);
  ```

  Protected copy constructor.

- ```
  virtual ~ccCaliperFinderBaseManualRunParams() = 0;
  ```

  Destructor.
  ```

Operators

operator=

```
ccCaliperFinderBaseRunParams& operator=(
  const ccCaliperFinderBaseRunParams& newObj);
```

Protected assignment operator. Prohibits copying and assignment by base pointers.

Parameters

newObj The new object that is a copy of this object.

Public Member Functions

setAffineSamplingAndCaliperRunParams

```
void setAffineSamplingAndCaliperRunParams (
  const cmStd vector<ccAffineSamplingParams>& affParams,
  const cmStd vector<ccCaliperRunParams>& clpParams);
```

Sets the affine sampling and caliper run-time parameters. These lists are used in manual mode where each affine parameter list item corresponds to the affine sampling parameters for one caliper and each caliper parameter list item corresponds to the run-time parameters for one caliper.

The defaults are empty vectors which are not valid for running a tool.

Parameters

affParams The affine sampling parameters list.

clpParams The caliper run-time parameters list.

Throws

ccCaliperFinderBaseDefs::BadParams

If **affParams.size()** is not equal to **clpParams.size()**,
or if **affParams.size()** < **minNumCalipers()**,
or if **clpParams.size()** < **minNumCalipers()**.

ccCaliperFinderBaseManualRunParams

ccCaliperFinderBaseResult

```
#include <ch_cvl/clpfind.h>

class ccCaliperFinderBaseResult : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This is an abstract base class for the Shape Finder tool results.

Constructors/Destructors

ccCaliperFinderBaseResult

```
ccCaliperFinderBaseResult();

ccCaliperFinderBaseResult(
  const ccCaliperFinderBaseResult&);

virtual ~ccCaliperFinderBaseResult();
```

- `ccCaliperFinderBaseResult();`

  Protected default constructor. **fitterResultsValid()** is set to false.

- ```
  ccCaliperFinderBaseResult(
    const ccCaliperFinderBaseResult&);
  ```

  Protected copy constructor.

- `virtual ~ccCaliperFinderBaseResult();`

  Destructor.
  ```

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseResult& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperFinderBaseResult** objects are considered equal if and only if their *fitterResultsValid*, *pose*, caliper results, and edge positions are equal.

Parameters

that The object to compare to this object.

operator=

```
ccCaliperFinderBaseResult& operator=(
  const ccCaliperFinderBaseResult& newObj);
```

Protected assignment operator. Prohibits copying and assignment by base pointers.

Parameters

newObj The new object that is a copy of this object.

Public Member Functions

found

```
virtual bool found() const;
```

Returns true for a found shape with a valid result. Otherwise returns false.

fitterResultsValid

```
bool fitterResultsValid() const;
```

Returns true if the fitter results are valid. Otherwise returns false.

pose

```
const cc2Xform& pose() const;
```

Returns the pose that maps the expected shape to the found shape.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the fitter results are invalid.

caliperResults

```
const cmStd vector<ccCaliperResultSet> &caliperResults();
```

Returns the caliper results.

edgePositions

```
const cmStd vector<cc2Vect> &edgePositions() const;
```

Returns the found edge positions.

caliperTime

```
double caliperTime() cons;
```

Returns the time (in seconds) spent running calipers.

totalTime

```
double totalTime() const;
```

Returns the total time (in seconds) to find the shape.

Protected Member Functions

checkFitterResults

```
void checkFitterResults() const;
```

Checks whether or not the fitter results are valid. If the results are valid, the function returns. If the results are not valid it throws the error below.

Throws

ccCaliperFinderBaseDefs::NotComputed

If the fitter results are invalid because they were not computed.

ccCaliperFinderBaseResult

ccCaliperFinderBaseRunParams

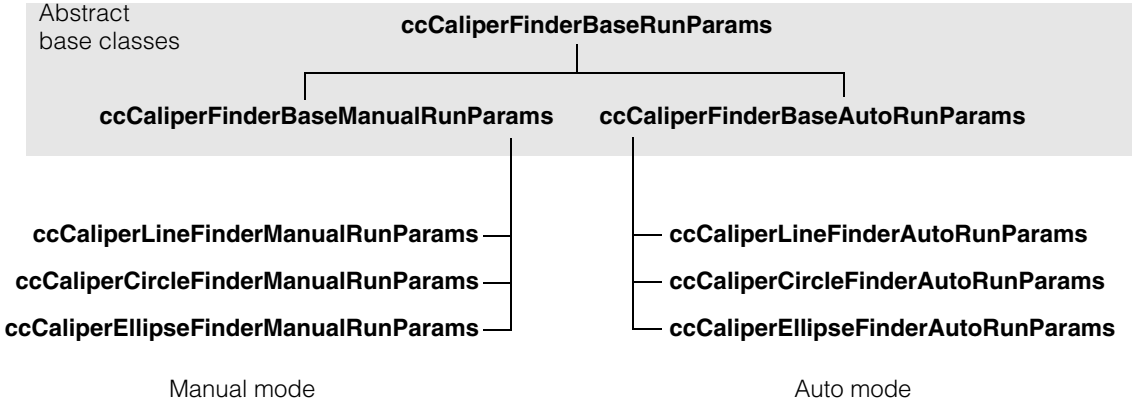
```
#include <ch_cvl/clpfind.h>

class ccCaliperFinderBaseRunParams : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This is an abstract base class for the Shape Finder tool run-time parameters. See the following class hierarchy diagram.



Constructors/Destructors

ccCaliperFinderBaseRunParams

```
ccCaliperFinderBaseRunParams(
  const cc2Xform& startPose,
  bool decrementNumIgnore);

ccCaliperFinderBaseRunParams(
  const ccCaliperFinderBaseRunParams&);

virtual ~ccCaliperFinderBaseRunParams() = 0;
```

- ```
  ccCaliperFinderBaseRunParams(
    const cc2Xform& startPose,
    bool decrementNumIgnore);
  ```

  Protected; constructs a finder base run-time parameters object.

  Parameters

  startPose A transform that maps the expected shape from shape space to its expected pose in the run-time client space. It also maps the caliper affine rectangles into client space.

  decrementNumIgnore When true, *numIgnore* in the shape fit parameters is decremented for each caliper that fails to find an edge. When false, no decrement occurs. The default is true.

  Throws

  ccCaliperFinderBaseDefs::BadParams If *startPose* is singular.

- ```
  ccCaliperFinderBaseRunParams(
    const ccCaliperFinderBaseRunParams&);
  ```

  Protected copy constructor.

- ```
  virtual ~ccCaliperFinderBaseRunParams() = 0;
  ```

  Destructor.
  ```

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperFinderBaseRunParams** objects are considered equal if, and only if, all their members are equal.

Parameters

that	The object to compare to this object.

operator=

```
ccCaliperFinderBaseRunParams& operator=(
  const ccCaliperFinderBaseRunParams& newObj);
```

Protected assignment operator. Prohibits copying and assignment by base pointers.

Parameters

newObj	The new object that is a copy of this object.

Public Member Functions

startPose

```
const cc2Xform& startPose() const;

void startPose(const cc2Xform& startPose);
```

A transform that maps the expected shape from shape space to its expected pose in the run-time client space. It also maps the caliper affine rectangles into client space.

The default is the identity transform.

- `const cc2Xform& startPose() const;`

  Returns the current start pose.

- `void startPose(const cc2Xform& startPose);`

  Sets a new start pose.

  Parameters

startPose	The new start pose.

Throws

ccCaliperFinderBaseDefs::BadParams
If *startPose* is singular.

decrementNumIgnore

```
bool decrementNumIgnore() const;

void decrementNumIgnore(bool ignore);
```

When true, *numIgnore* in the shape fit parameters is decremented for each caliper that fails to find an edge. When false, no decrement occurs. The default is true.

Notes

An internal copy is decremented not the actual fitter parameters given by the user.

- `bool decrementNumIgnore() const;`

  Returns the current setting, true or false.

- `void decrementNumIgnore(bool ignore);`

  Sets the ignore flag.

  Parameters

  ignore The new flag, true or false.

affineSamplingParamsList

```
virtual const cmStd
vector<ccAffineSamplingParams> affineSamplingParamsList(
  const cc2XformBase &clientFromImage =
                                    cc2XformLinear()) const;
```

Parameters

clientFromImage
The client space to image space transform.

Returns the affine sampling parameters list. This list is used in manual mode where each list item corresponds to the affine sampling parameters for one caliper.

Notes

These sampling parameters do not reflect the start pose.

If you call this function on a default constructed object, it returns an empty vector which is not valid for running a tool. You must first set valid values for caliper size, number of calipers, and caliper sampling.

caliperRunParamsList

```
const cmStd vector<ccCaliperRunParams>
                                &caliperRunParamsList() const;
```

Returns the caliper run-time parameters list. This list is used in manual mode where each list item corresponds to the run-time parameters for one caliper.

Notes

The default is an empty vector which is not valid for running a tool.

minNumCalipers

```
virtual c_Int16 minNumCalipers() const = 0;
```

Returns the minimum number of calipers required.

Protected Member Functions

affineSamplingParamsList

```
void affineSamplingParamsList (
  const cmStd vector<ccAffineSamplingParams>&
                                    affineSamplingParamsList);
```

Sets the affine sampling parameters list. This list is used in manual mode where each list item corresponds to the affine sampling parameters for one caliper.

Parameters

affineSamplingParamsList
The new list.

caliperRunParamsList

```
void caliperRunParamsList (
  const cmStd vector<ccCaliperRunParams>&
                                        caliperRunParamsList);
```

Sets the caliper run-time parameters list. This list is used in manual mode where each list item corresponds to the run-time parameters for one caliper.

Parameters

caliperRunParamsList
The new list.

ccCaliperFinderBaseRunParams

ccCaliperLineFinderAutoRunParams

```
#include <ch_cvl/clpfind.h>

class ccCaliperLineFinderAutoRunParams :
                        public ccCaliperFinderBaseAutoRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Encapsulates the Line Finder tool run-time parameters for running the tool in *auto* mode.

Constructors/Destructors

Notes

Default constructed objects do not contain valid run-time parameters.

ccCaliperLineFinderAutoRunParams

```
ccCaliperLineFinderAutoRunParams (
  const ccLineSeg& expectedLine = ccLineSeg(),
  c_Int16 numCalipers = 0,
  const ccDPair& caliperSize = ccDPair(0, 0),
  const ccCaliperRunParams& caliperRunParams =
                                      ccCaliperRunParams(),
  const ccDPair& caliperSampling = ccDPair(1, 1),
  const ccRadian& skew = ccRadian(0),
  ccAffineSamplingParams::Interpolation
                  interpolationMethod =
                  ccAffineSamplingParams::eBilinear,
  const ccLineFitParams& lineFitParams = ccLineFitParams(),
  const cc2Xform& startPose = cc2Xform(),
  bool decrementNumIgnore = true);

ccCaliperLineFinderAutoRunParams (
  const ccAffineRectangle &affineRectangle,
  c_Int16 numCalipers = 0,
  double caliperHeight = 0,
  const ccCaliperRunParams &caliperRunParams =
                                      ccCaliperRunParams(),
  const ccDPair &caliperSampling = ccDPair(1, 1),
  ccAffineSamplingParams::Interpolation
                        interpolationMethod =
```

```
                                ccAffineSamplingParams::eBilinear,
  const ccLineFitParams& lineFitParams = ccLineFitParams(),
  const cc2Xform& startPose = cc2Xform(),
  bool decrementNumIgnore = true);

virtual ~ccCaliperLineFinderAutoRunParams() {}
```

- ```
  ccCaliperLineFinderAutoRunParams (
    const ccLineSeg& expectedLine = ccLineSeg(),
    c_Int16 numCalipers = 0,
    const ccDPair& caliperSize = ccDPair(0, 0),
    const ccCaliperRunParams& caliperRunParams =
                                          ccCaliperRunParams(),
    const ccDPair& caliperSampling = ccDPair(1, 1),
    const ccRadian& skew = ccRadian(0),
    ccAffineSamplingParams::Interpolation
                    interpolationMethod =
                    ccAffineSamplingParams::eBilinear,
    const ccLineFitParams& lineFitParams = ccLineFitParams(),
    const cc2Xform& startPose = cc2Xform(),
    bool decrementNumIgnore = true);
  ```

  Constructs a line finder auto run parameters object where the expected line is defined by a **ccLineSeg** object.

  Parameters

  expectedLine The line you expect the tool to find in images.

  numCalipers The number of calipers the tool should use to find the line.

  caliperSize The caliper size, width and height in client coordinates. All calipers are the same size.

  caliperRunParams
  The run-time parameters object to be used when the Caliper tool is run.

  caliperSampling The sampling rate in samples/pixel for x and y, to be used on the calipers. For example, the following calculates the number of caliper samples in the x direction:

  If xSize is **caliperSize.x()** in image coordinates, then the effective number of samples in the x direction will be -

  c_Int16(xSize * **caliperSampling.x()** + 0.5)
  ```

skew A skew angle. Normally calipers are perpendicular to the expected line. You may wish to skew the calipers to avoid features in the image that could be confused with the line. See *skew* on page 854.

interpolationMethod
The interpolation method used with the calipers. Must be one of the **ccAffineSamplingParams::Interpolation** enums. The default is *eBilinear*.

See the **ccAffineSamplingParams** reference page.

lineFitParams The run-time parameters to use with the Fitting tool.

startPose A transform that maps the expected line from line space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *ccLineFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

Throws

ccCaliperFinderBaseDefs::BadParams
If *numCalipers* is not zero and is less than **minNumCalipers()**,
or if *caliperSize* is less than (0, 0),
or if *caliperSampling* is less than or equal to (0, 0),
or if *skew* is +90° or -90°,
or if *startPose* is singular.

- ```
  ccCaliperLineFinderAutoRunParams (
    const ccAffineRectangle &affineRectangle,
    c_Int16 numCalipers = 0,
    double caliperHeight = 0,
    const ccCaliperRunParams &caliperRunParams =
                                        ccCaliperRunParams(),
    const ccDPair &caliperSampling = ccDPair(1, 1),
    ccAffineSamplingParams::Interpolation
                          interpolationMethod =
                          ccAffineSamplingParams::eBilinear,
    const ccLineFitParams& lineFitParams = ccLineFitParams(),
    const cc2Xform& startPose = cc2Xform(),
    bool decrementNumIgnore = true);
  ```

  Constructs a line finder auto run parameters object where the expected line is defined by an affine rectangle. Specifying the affine rectangle also sets *caliperWidth* and the skew.
  ```

Parameters

affineRectangle A rectangle that defines the expected line segment.

numCalipers The number of calipers the tool should use to find the line.

caliperHeight The caliper height in pixels. The caliper width is defined by the *affineRectangle* width. All calipers are the same size.

caliperRunParams
The run-time parameters object to be used when the Caliper tool is run.

caliperSampling The sampling rate in samples/pixel for x and y, to be used on the calipers. For example, the following calculates the number of caliper samples in the x direction:

If xSize is **caliperSize.x()** in image coordinates, then the effective number of samples in the x direction will be -

c_Int16(xSize * **caliperSampling.x()** + 0.5)

interpolationMethod
The interpolation method used with the calipers. Must be one of the **ccAffineSamplingParams::Interpolation** enums. The default is *eBilinear*.

See the **ccAffineSamplingParams** reference page.

lineFitParams The run-time parameters to use with the Fitting tool.

startPose A transform that maps the expected line from line space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *ccLineFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

Throws

ccCaliperFinderBaseDefs::BadParams
If *numCalipers* is not zero and is less than **minNumCalipers()**,
or if **affineRectangle.degen()** is true,
or if *caliperHeight* is less than 0,
or if *caliperSampling* is less than or equal to (0, 0),
or if *startPose* is singular.

- `virtual ~ccCaliperLineFinderAutoRunParams() {}`

  Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperLineFinderAutoRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperLineFinderAutoRunParams** object to compare to this object.

Public Member Functions

expectedLine

```
const ccLineSeg &expectedLine() const;

void expectedLine(const ccLineSeg &expectedLine);
```

- `const ccLineSeg &expectedLine() const;`

  Returns the expected line segment.

- `void expectedLine(const ccLineSeg &expectedLine);`

  Sets the expected line segment.

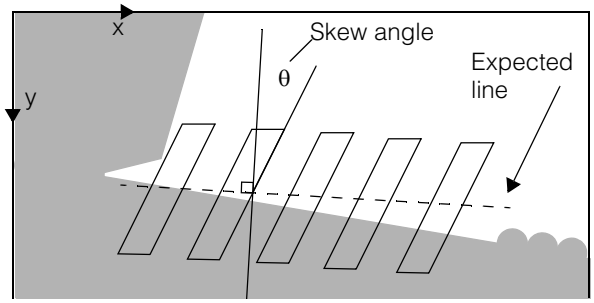
  Throws

  ccCaliperFinderBaseDefs::BadParams
  If **expectedLine.p1()** == **expectedLine.p2()**.

skew

```
const ccRadian &skew() const;
void skew(const ccRadian &skew);
```

When skew is 0°, calipers are perpendicular to the expected line. You may wish to skew the calipers to avoid features in the image that could be confused with the line. See the following example.



- `const ccRadian &skew() const;`

  Returns the current skew angle. The default is 0°.

- `void skew(const ccRadian &skew);`

  Sets a new skew angle.

  Parameters

  skew The new skew angle.

  Throws

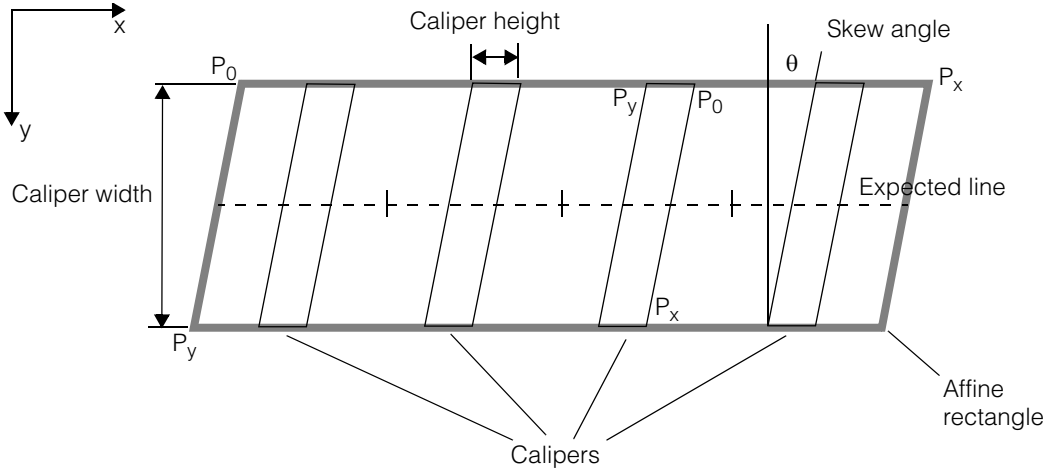
  ccCaliperFinderBaseDefs::BadParams
  If f *skew* is +90° or -90°.

affineRectangle

```
void affineRectangle(
  const ccAffineRectangle &affineRectangle);

ccAffineRectangle affineRectangle() const;
```

You can use an affine rectangle to define the expected line, caliper width, and skew. See the example below.



The expected line bisects the rectangle as shown, and the calipers are evenly distributed along the expected line.

- `ccAffineRectangle affineRectangle() const;`

  Returns the current affine rectangle.

- `void affineRectangle(`
  `const ccAffineRectangle &affineRectangle);`

  Sets a new affine rectangle.

  Parameters

  affineRectangle The new affine rectangle.

  Throws

  ccCaliperFinderBaseDefs::BadParams
  If **affineRectangle.degen()** is true.

  ccCaliperFinderBaseDefs::BadParams if affineRectangle.degen().

ccCaliperLineFinderAutoRunParams

lineFitParams

```
const ccLineFitParams &lineFitParams() const;
void lineFitParams(const ccLineFitParams &lineFitParams);
```

- `const ccLineFitParams &lineFitParams() const;`

  Returns the line fit parameters.

- `void lineFitParams(const ccLineFitParams &lineFitParams);`

  Sets new line fit parameters.

  Parameters

  lineFitParams The new parameters.

minNumCalipers

```
virtual c_Int16 minNumCalipers() const;
```

Returns the minimum number of calipers required (2).

ccCaliperLineFinderManualRunParams

```
#include <ch_cvl/clpfind.h>

class ccCaliperLineFinderManualRunParams :
                      public ccCaliperFinderBaseManualRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Encapsulates the Line Finder tool run-time parameters for running the tool in *manual* mode.

Constructors/Destructors

ccCaliperLineFinderManualRunParams

```
ccCaliperLineFinderManualRunParams (
  const ccLineSeg& expectedLine = ccLineSeg(),
  const cmStd vector<ccAffineSamplingParams>& affParams =
                    cmStd vector<ccAffineSamplingParams>(),
  const cmStd vector<ccCaliperRunParams>& clpParams =
                          cmStd vector<ccCaliperRunParams>(),
  const ccLineFitParams& lineFitParams = ccLineFitParams(),
  const cc2Xform &startPose = cc2Xform(),
  bool decrementNumIgnore = true);

virtual ~ccCaliperLineFinderManualRunParams() {}
```

- ```
  ccCaliperLineFinderManualRunParams (
    const ccLineSeg& expectedLine = ccLineSeg(),
    const cmStd vector<ccAffineSamplingParams>& affParams =
                      cmStd vector<ccAffineSamplingParams>(),
    const cmStd vector<ccCaliperRunParams>& clpParams =
                          cmStd vector<ccCaliperRunParams>(),
    const ccLineFitParams& lineFitParams = ccLineFitParams(),
    const cc2Xform &startPose = cc2Xform(),
    bool decrementNumIgnore = true);
  ```

  Constructs a line finder manual run-time parameters object.
  ```

Notes

Default constructed objects do not contain valid run-time parameters.

Parameters

expectedLine The line you expect the tool to find in images.

affParams A vector of affine rectangle sampling parameters, one parameter set for each caliper.

clpParams A vector of caliper run-time parameters, one parameter set for each caliper.

lineFitParams The run-time parameters to use with the Fitting tool.

startPose A transform that maps the expected line from line space to its expected pose in the run-time client space.

decrementNumIgnore
When true, *ccLineFitParams::numIgnore* is decremented for each caliper that fails to find an edge. When false, no decrement occurs.

Throws

ccCaliperFinderBaseDefs::BadParams
If **affParams.size()** is not equal to **clpParams.size()**, or if params sizes are not zero and less than **minNumCalipers()**, or if *startPose* is singular.

- `virtual ~ccCaliperLineFinderManualRunParams() {}`

Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseRunParams& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperLineFinderManualRunParams** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperLineFinderManualRunParams** object to compare to this object.

Public Member Functions

expectedLine

```
const ccLineSeg &expectedLine() const;
void expectedLine(const ccLineSeg &expectedLine);
```

The line segment you expect to find in images processed by the Line Finder tool. The expected line should be close to actual lines in the images.

- `const ccLineSeg &expectedLine() const;`

  Returns the expected line segment.

- `void expectedLine(const ccLineSeg &expectedLine);`

  Sets the expected line segment.

  Throws
  ccCaliperFinderBaseDefs::BadParams
  If **expectedLine.p1()** == **expectedLine.p2()**.

lineFitParams

```
const ccLineFitParams &lineFitParams() const;
void lineFitParams(const ccLineFitParams &lineFitParams);
```

- `const ccLineFitParams &lineFitParams() const;`

  Returns the line fitter run-time parameters.

- `void lineFitParams(const ccLineFitParams &lineFitParams);`

  Sets new line fitter run-time parameters.

  Parameters
  lineFitParams The new parameters.

minNumCalipers

```
virtual c_Int16 minNumCalipers() const;
```

Returns the minimum number of calipers required (2).

ccCaliperLineFinderManualRunParams

ccCaliperLineFinderResult

```
#include <ch_cvl/clpfind.h>

class ccCaliperLineFinderResult :
                                public ccCaliperFinderBaseResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the results from the Line Finder tool.

Constructors/Destructors

ccCaliperLineFinderResult

```
ccCaliperLineFinderResult() {};

virtual ~ccCaliperLineFinderResult() {}
```

- `ccCaliperLineFinderResult() {};`

  Default constructor. Creates a default constructed (unfound) line result object.

- `virtual ~ccCaliperLineFinderResult() {}`

  Destructor.

Operators

operator==

```
virtual bool operator==(
  const ccCaliperFinderBaseResult& that) const;
```

Compares this object to another object of the same type. Returns true if this object equals *that*. Returns false otherwise.

Two **ccCaliperLineFinderResult** objects are considered equal if, and only if, all their members and base classes are equal.

Parameters

that The **ccCaliperLineFinderResult** object to compare to this object.

Public Member Functions

found

```
virtual bool found() const;
```

Returns true if a line was found. Returns false otherwise.

A line is found if its computed error is less than the error *threshold* specified in the Line Fitting tool parameters. See **ccLineFitParams::threshold()**.

lineFit

```
const ccLineFitResults &lineFit() const;
```

Returns the line fitting result.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

line

```
const ccFLine &line() const;
```

Returns the computed line.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

lineSeg

```
const ccLineSeg &lineSeg() const;
```

Returns the best-fit line segment of the found line (the found line is an infinite length line). The best-fit line segment center is aligned with the expected line center and is the same length as the expected line.

Throws

ccCaliperFinderBaseDefs::NotComputed
If the Fitting tool results are invalid.

lineFitTime

```
double lineFitTime() const;
```

Returns the line fitting time, in seconds.

ccCaliperOneResult

```
#include <ch_cvl/caliper.h>

class ccCaliperOneResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single edge or edge pair result produced by the Caliper tool.

Note You should not construct a **ccCaliperOneResult** directly.

Constructors/Destructors

ccCaliperOneResult

```
ccCaliperOneResult ();
```

Do not construct **ccCaliperOneResult**s directly.

Public Member Functions

position

```
double position () const;
```

Returns the position at which this edge or edge pair was found. The returned position is a signed value indicating the distance between the center of the projection region (or another user-specified origin) and the origin of the edge model.

Notes

The position of an edge pair is computed so that the midpoint between the desired edges is aligned with the midpoint of the result edges.

score

```
double score () const;
```

Returns the overall score that this edge or edge pair received. The score is between 0.0 and 1.0. This is the geometric mean of all scores computed for the individual **ccCaliperScore** scoring methods.

resultEdges

```
const cmStd vector<ccCaliperResultEdge>& resultEdges ()
  const;
```

Returns a vector containing each of the individual **ccCaliperOneResult** edges that contribute to this result. They are listed in order of increasing position.

scores

```
const cmStd vector<double>& scores () const;
```

Returns a vector containing each of the individual scores contributing to this result. The order of the scores is the same as the order of the vector of **ccCaliperScore** scoring methods you specified when calling **ccCaliperRunParams()::scoringMethods()**.

draw

```
void draw(
  ccGraphicList& graphList,
  const ccCaliperBaseResultSet& set,
  c_UInt32 drawMode=ccCaliperDefs::eDrawStandard,
  const ccCvlString& label=ccCvlString(),
  double plotHeight=ccCaliperDefs::kDefaultPlotHeight)
                                                    const;
```

Appends result graphics for this **ccCaliperOneResult** to the supplied **ccGraphicList**.

Parameters

graphList — The graphics list to append the graphics to.

set — You must supply a reference to a **ccCaliperBaseResultSet** that contains this **ccCaliperOneResult**.

drawMode — The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccCaliperDefs::eDrawEdgeLine
ccCaliperDefs::eDrawLabel
ccCaliperDefs::eDrawProjFilter
ccCaliperDefs::eDrawProjectionRegion
ccCaliperDefs::eDrawStandard

label — If you include *ccCaliperDefs::eDrawLabel* in *drawMode*, then the contents of *label* are used to label the center of mass.

plotHeight — The height of the projection and filter plots, in client coordinates.

ccCaliperProjectionParams

```
#include <ch_cvl/caliper.h>

class ccCaliperProjectionParams: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

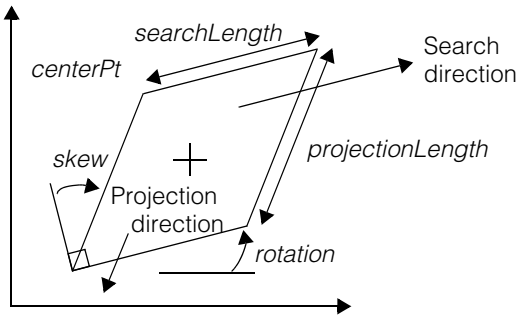
In general, you should not use this class to specify projection parameters. Instead, specify the projection region as a **ccAffineRectangle** and **ccAffineRectangleSamplingParams**.

Constructors/Destructors

ccCaliperProjectionParams

```
ccCaliperProjectionParams (const cc2Vect& centerPt,
  double searchLength, double projectionLength,
  const ccRadian& rotation =
  ccRadian(ccCaliperDefs::kDefaultRotation),
  const ccRadian& skew =
  ccRadian(ccCaliperDefs::kDefaultSkew),
  ccCaliperDefs::Interpolation method =
  ccCaliperDefs::kDefaultInterpolation);
```

Constructs a **ccCaliperProjectionParams** using the supplied parameters. The following diagram shows how the supplied parameters define the projection region:



Parameters

centerPt The center point of the projection region, specified in the client coordinate system of the input image.

searchLength The size of the projection region in the direction perpendicular to the projection direction, in client coordinates. The Caliper tool will convert this quantity into image coordinates and round it to the next higher integer value to determine the number of rays in the projection region. This in turn determines the dimensions of the projection image.

projectionLength The size of the projection region in the projection direction, in client coordinates. The Caliper tool will convert this quantity into image coordinates and round it to the next highest value to determine the number of interpolated pixels each ray.

rotation The rotation angle. This is defined as the angle from the client coordinate system x-axis to the search direction side of the projection region.

skew The skew angle. This is defined as the angle from a line perpendicular to the base of the projection region to the side of the projection region.

rotation plus *skew* is equal to the angle of the projection direction with respect to the client coordinate system y-axis.

method The sampling method to use to determine the values to sum to form the projection image. *method* must be one of the following values:

ccCaliperDefs::eNone
ccCaliperDefs::eBilinear

Public Member Functions

windowCenter

```
cc2Vect windowCenter () const;

void windowCenter (const cc2Vect& point);
```

- `cc2Vect windowCenter () const;`

  Returns the center point of the projection region in the input image client coordinate system. If you supply a **ccCaliperProjectionParams** as input to **cfCaliperRun**, then all positions reported by the tool are relative to this center point.

- `void windowCenter (const cc2Vect& point);`

  Sets the center point of the projection region in the input image client coordinate system. If you supply a **ccCaliperProjectionParams** as input to **cfCaliperRun**, then all positions reported by the tool are relative to this center point.

  Parameters

  point The center point to set.

windowSearchLength

```
double windowSearchLength () const;

void windowSearchLength (double length);
```

- `double windowSearchLength () const;`

  Returns the search length of the projection region in client coordinate system units.

- `void windowSearchLength (double length);`

  Sets the search length of the projection region in client coordinate system units. The search length specifies the size of the projection region in the search direction.

  Parameters

  length The length to set. *length* must be greater than 0.

  Throws

  ccCaliperDefs::BadParams

  length is less than or equal to zero.

  Notes

  The search length value is converted to input image coordinates, then rounded up to the next integral value, to determine the number of rays in the projection region and the number of pixels in the projection image.

windowProjectionLength

```
double windowProjectionLength () const;

void windowProjectionLength (double length);
```

- `double windowProjectionLength () const;`

  Returns the projection length of the projection region in client coordinate system units.

- `void windowProjectionLength (double length);`

  Sets the projection length of the projection region in client coordinate units. The projection length specifies the size of the window in the projection direction.

  Parameters

  length The length to set. *length* must be greater than 0.

  Throws

  ccCaliperDefs::BadParams
  length is less than or equal to zero.

windowRotation

```
ccRadian windowRotation () const;
void windowRotation (const ccRadian& angle);
```

- `ccRadian windowRotation () const;`

  Returns the rotation angle for the projection region. The rotation angle is the angle from the client coordinate system x-axis to the search direction side of the projection region.

- `void windowRotation (const ccRadian& angle);`

  Sets the rotation angle for the projection region. The rotation angle is the angle from the client coordinate system x-axis to the search direction side of the projection region. If this angle is 0, the search direction is parallel to the x-axis and points toward increasing x. If the angle is 90 degrees, the search direction is parallel to the y-axis and points toward increasing y.

  Parameters

  angle The angle to set

windowSkew

```
ccRadian windowSkew () const;
void windowSkew (const ccRadian& angle);
```

- `ccRadian windowSkew () const;`

  Returns the skew angle for the projection region. The skew angle is the angle from a line drawn perpendicular to the search direction to the projection direction. The skew angle plus the rotation angle gives the projection direction, relative to the client coordinate system x-axis.

- `void windowSkew (const ccRadian& angle);`

  Sets the skew angle for the projection region. The skew angle is the angle from a line drawn perpendicular to the search direction to the projection direction. The skew angle plus the rotation angle gives the projection direction, relative to the client coordinate system x-axis.

  Parameters

  angle The angle to set

  Throws

  ccCaliperDefs::BadParams
  angle is greater than $\pi/2$ radians or less than $-\pi/2$ radians.

interpolation

```
ccCaliperDefs::Interpolation interpolation () const;
void interpolation (ccCaliperDefs::Interpolation method);
```

- `ccCaliperDefs::Interpolation interpolation () const;`

  Returns the sampling method specified for this **ccCaliperProjectionParams**. This function returns one of the following values:

  ccCaliperDefs::eNone
  ccCaliperDefs::eBilinear

- `void interpolation (ccCaliperDefs::Interpolation method);`

  Sets the sampling method specified for this **ccCaliperProjectionParams**.

  Parameters

  method The sampling method to set. *method* must be one of the following values:

  ccCaliperDefs::eNone
  ccCaliperDefs::eBilinear

ccCaliperProjectionParams

ccCaliperResultEdge

```
#include <ch_cvl/caliper.h>

class ccCaliperResultEdge;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single edge located by the Caliper tool in the input image.

Caution *You should not attempt to construct a* **ccCaliperResultEdge** *yourself, and you should not attempt to call any member function that sets a value within a* **ccCaliperResultEdge***.*

Constructors/Destructors

ccCaliperResultEdge

```
ccCaliperResultEdge (double position = 0.0,
  double contrast = 0.0);
```

Construct this **ccCaliperResultEdge** with the given parameters.

Parameters

position	The location of the edge.
contrast	The contrast of the edge.

Public Member Functions

position

```
double position () const;
void position (double pos);
```

- `double position () const;`

  Returns the position at which this edge was found. The returned value is a signed quantity indicating the distance from the center of the caliper window (or other user-supplied origin) and the edge, in client coordinate system units.

- `void position (double pos);`

  This function is for internal use only. Do not call this function.

  Parameters

  pos The position to set

contrast

```
double contrast () const;
void contrast (double con);
```

- `double contrast () const;`

  Returns the contrast of this edge. The contrast is expressed as the difference in normalized grey level between the pixels on the two sides of the edge.

- `void contrast (double con);`

  This function is for internal use only. Do not call this function.

  Parameters

  con The contrast to set

ccCaliperResultSet

```
#include <ch_cvl/caliper.h>

class ccCaliperResultSet: public ccCaliperBaseResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the results returned by a single application of the Caliper tool to an input image in edge mode. Results common to both modes can be found in **ccCaliperBaseResultSet**.

Constructors/Destructors

ccCaliperResultSet

```
ccCaliperResultSet ();
```

Constructs **ccCaliperResultSet** with no result information.

Public Member Functions

edgeDetectTime

```
double edgeDetectTime () const;
```

Returns the amount of time, in seconds, required to locate all the edges in the image.

resultEdges

```
const cmStd vector<ccCaliperResultEdge>& resultEdges ()
  const;
```

Returns a vector of **ccCaliperResultEdge**s containing all the edges from the projection image. The edges are returned in order of increasing position in the search direction.

draw

```
void draw(
  ccGraphicList& graphList,
  c_UInt32 drawMode = ccCaliperDefs::eDrawStandard,
  double plotHeight = ccCaliperDefs::kDefaultPlotHeight)
                                                    const;
```

Appends result graphics for all the results in this **ccCaliperResultSet** to the supplied **ccGraphicList**.

Parameters

graphList	The graphics list to append the graphics to.
drawMode	The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values: *ccCaliperDefs::eDrawEdgeLine* *ccCaliperDefs::eDrawLabel* *ccCaliperDefs::eDrawProjFilter* *ccCaliperDefs::eDrawProjectionRegion* *ccCaliperDefs::eDrawStandard* *ccCaliperDefs::eDrawRawEdges*
plotHeight	The height of the projection and filter plots, in client coordinates.

ccCaliperRunParams

```
#include <ch_cvl/caliper.h>

class ccCaliperRunParams: public ccCaliperBaseRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the run-time parameters for the Caliper tool used in edge mode, including the definition of the expected edge or edge pair and the scoring methods to apply to candidate edges in the input image.

Constructors/Destructors

ccCaliperRunParams

```
ccCaliperRunParams (double edgePosition = 0.0,
  ceEdgePolarity edgePolarity = ceDontCare);

ccCaliperRunParams (double edge1Position,
  ceEdgePolarity edge1Polarity, double edge2Position,
  ceEdgePolarity edge2Polarity);
```

- ```
  ccCaliperRunParams (double edgePosition = 0.0,
    ceEdgePolarity edgePolarity = ceDontCare);
  ```

  Constructs a **ccCaliperRunParams** to search for a single-edge model.

  Parameters

  edgePosition The position of the edge with respect to the edge model origin.

  edgePolarity The polarity of the edge, defined as the polarity in the search direction. *edgePolarity* must be one of the following values:

  ceDarkToLight
  ceLightToDark
  ceDontCare

  A value of *ceDontCare* means the edge can have either polarity.
  ```

- ```
  ccCaliperRunParams (double edge1Position,
    ceEdgePolarity edge1Polarity,
    double edge2Position, ceEdgePolarity edge2Polarity);
  ```

  Constructs a **ccCaliperRunParams** to search for an edge-pair edge model.

  Parameters

  edge1Position — The position of the first edge with respect to the edge model origin.

  edge1Polarity — The polarity of the first edge, defined as the polarity in the search direction. *edge1Polarity* must be one of the following values:

  ceDarkToLight
  ceLightToDark
  ceDontCare

  A value of *ceDontCare* means the edge can have either polarity.

  edge2Position — The position of the second edge with respect to the edge model origin.

  edge2Polarity — The polarity of the second edge, defined as the polarity in the search direction. *edge2Polarity* must be one of the following values:

  ceDarkToLight
  ceLightToDark
  ceDontCare

  A value of *ceDontCare* means the edge can have either polarity.

Public Member Functions

filterHalfSize

```
c_Int16 filterHalfSize () const;

void filterHalfSize (c_Int16 halfSize);
```

- ```
  c_Int16 filterHalfSize () const;
  ```

  Returns the half-size, in units of the projection image's image coordinate system, of the filter defined for this **ccCaliperRunParams**.

- `void filterHalfSize (c_Int16 halfSize);`

  Sets the half-size, in units of the projection image's image coordinate system, of the filter defined for this **ccCaliperRunParams**. The filter is constructed by composing a Gaussian curve with a σ equal to one fourth of the specified half size, computing the first derivative of the curve, then taking its negative.

  Parameters

  halfSize The half-size of the filter. The resulting filter kernel is 2**halfSize*+1 pixels in size.

  Notes

  halfSize is specified in pixels defined by the projection region, not client coordinate units.

  Throws

  ccCaliperDefs::BadParams
  halfSize is less than 1.

contrastThreshold

```
double contrastThreshold () const;

void contrastThreshold (double thresh);
```

- `double contrastThreshold () const;`

  Returns the contrast threshold for 1-D edge detection. Any result edge whose contrast value is less than the contrast threshold is ignored. The threshold is specified as the difference in normalized pixel values on the two sides of the edge.

- `void contrastThreshold (double thresh);`

  Sets the contrast threshold for 1-D edge detection. Any result edge whose contrast value is less than the contrast threshold is ignored. The threshold is specified as the difference in normalized pixel values on the two sides of the edge.

  Parameters

  thresh The threshold to set

maxNumResults

```
c_Int16 maxNumResults () const;

void maxNumResults (c_Int16 num);
```

- `c_Int16 maxNumResults () const;`

  Returns the maximum number of results that the tool returns.

- `void maxNumResults (c_Int16 num);`

  Sets the maximum number of results that the tool returns.

  Parameters

  num The maximum number of results to return

  Notes

  If you specify 0 for *num,* the tool performs the projection, filtering, and edge detection steps, but does not produce any scored results.

  When using the Caliper tool on a multiprocessor PC, use the **ccCaliperRunParams::maxNumResults()** setter with the maximum number of expected individual results as its argument to reduce memory contention between multiple threads running on different processors. This method results in improved Caliper performance on a multiprocessor PC over that expected of a single-processor PC. This setting is necessary to improve performance on a multiprocessor PC, as *maxNumResults* is set to one by default.

  Throws

  ccCaliperDefs::BadParams

  num is less than 0.

scoringMethods

```
const cmStd vector<ccCaliperScore*> scoringMethods ()
  const;

void scoringMethods (
  const cmStd vector<ccCaliperScore*>& methods);
```

- `const cmStd vector<ccCaliperScore*> scoringMethods ()
  const;`

  Returns a vector of **ccCaliperScore** objects that define how edge candidates are scored.

- ```
  void scoringMethods (
    const cmStd vector<ccCaliperScore*>& methods);
  ```

  Sets the list of **ccCaliperScore** objects that determine how edge candidates are scored. Any existing list of **ccCaliperScore** objects is discarded.

  Parameters

  methods The scoring methods to set

  Notes

  If *methods* contains no objects (has a length of 0), then any attempt to run the Caliper tool with a value for the maximum number of results other than 0 will throw an error.

  A default-constructed **ccCaliperRunParams** object contains a single **ccScoreContrast** scoring method.

  Throws

  ccCaliperDefs::BadParams

  One or more of the **ccCaliperScore** objects in *method* cannot produce a score for the number of edges specified when you constructed this **ccCaliperRunParams**.

  You can determine which element of *methods* is causing the problem by calling the **ccCaliperScore::willScore()** function.

desiredEdges

```
const cmStd vector<ccCaliperDesiredEdge>& desiredEdges ()
  const;
```

Returns the model edges that you specified when you constructed this **ccCaliperRunParams**. If you specified a single edge, the returned vector has a size of 1; if you specified an edge pair, the returned vector has a size of 2.
```

ccCaliperRunParams

ccCaliperScanParams

```
#include <ch_cvl/caliper.h>

class ccCaliperScanParams: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the run-time parameters for operating the Caliper tool using scan mode. Scan mode automatically rotates the projection window about the application point to determine the orientation that produces the highest overall score.

Constructors/Destructors

ccCaliperScanParams

```
ccCaliperScanParams();

ccCaliperScanParams(ccRadian start, ccRadian end,
  ccRadian incr, bool interpolate, bool enable);
```

- `ccCaliperScanParams(};`

  Construct a **ccCaliperScanParams** with all parameters set to 0 or false.

- `ccCaliperScanParams(ccRadian start, ccRadian end, ccRadian incr, bool interpolate, bool enable);`

  Construct a **ccCaliperScanParams** initialized to the supplied values. For more information on the individual parameters, see the member functions in this class

  Parameters

start	The start of the range of angles to be scanned. The angle is from the client coordinate system x-axis to the projection direction.
end	The end of the range of angles to be scanned. The angle is from the client coordinate system x-axis to the projection direction.
incr	The amount by which to increment the angle between scans.

interpolate If true, intermediate angles are evaluated.

enable If true, scan mode is enabled for this object. If false, scan mode is disabled.

Operators

operator==

```
bool operator== (const ccCaliperScanParams& that) const;
```

Two **ccCaliperScanParams** objects are considered equal if their corresponding start angles, end angles, angle increments, interpolation settings, and enabled settings are the same.

Public Member Functions

scanStart

```
const ccRadian& scanStart() const;
```

```
void scanStart(const ccRadian& angle);
```

- `const ccRadian& scanStart() const;`

  Returns the start of the range of angles to be scanned.

- `void scanStart(const ccRadian& angle);`

  Sets the start of the range of angles to be scanned.

  Parameters

  angle The start angle measured from the client coordinate system x-axis to the projection direction.

scanEnd

```
const ccRadian& scanEnd() const;
```

```
void scanEnd(const ccRadian& angle);
```

- `const ccRadian& scanEnd() const;`

  Returns the end of the range of angles to be scanned.

- `void scanEnd(const ccRadian& angle);`

  Sets the end of the range of angles to be scanned.

Parameters

angle The end angle measured from the client coordinate system x-axis to the projection direction.

scanIncrement

```
const ccRadian& scanIncrement() const;
void scanIncrement(const ccRadian& angle);
```

- `const ccRadian& scanIncrement() const;`

  Returns the amount by which to increment the projection direction between scans.

- `void scanIncrement(const ccRadian& angle);`

  Sets the amount by which to increment the projection direction between scans.

  Parameters

  angle The increment value.

scanInterpol

```
bool scanInterpol() const;
void scanInterpol(bool interpol);
```

- `bool scanInterpol() const;`

  Returns whether or not this **ccCaliperScanParams** is configured to evaluate angles between the specified angle increments.

  `void scanInterpol(bool interpol);`

  Sets whether or not this **ccCaliperScanParams** is configured to evaluate angles between the specified angle increments. If interpolation is enabled, the Caliper tool will evaluate scan angles between the highest scoring angles specified by the start, stop, and increment values.

  Enabling interpolation increases the execution time required by the tool.

  Parameters

  interpol If true, interpolation is enabled. If false, interpolation is disabled.

ccCaliperScanParams

scanEnable

```
bool scanEnable() const;
void scanEnable(bool on);
```

- `bool scanEnable() const;`

  Returns true if this **ccCaliperScanParams** has scan mode enabled, false if scan mode is disabled.

- `void scanEnable(bool on);`

  Enables or disables scan mode for this **ccCaliperScanParams**. Since a single **ccCaliperRunParams** or **ccCaliperCorrelationRunParams** (both of which contain a **ccCaliperScanParams** object) can be used for multiple calipers, this function lets you enable or disable scan mode easily.

  Parameters

on	If true, scan mode is enabled. If false, scan mode is disabled.

ccCaliperScore

```
#include <ch_cvl/clpscore.h>

class ccCaliperScore : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	By Cognex-supplied classes only
Archiveable	Complex

A class from which intermediate classes such as **ccScoreOneSided** and **ccScoreTwoSided** are derived.

Note You can neither instantiate nor derive from this class. You should be familiar with the public member functions in this class; they are implemented in Cognex-supplied classes derived from this class.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const = 0;
```

Returns true if this **ccCaliperScore** is equivalent to the supplied **ccCaliperScore**.

Parameters

that The **ccCaliperScore** to compare to this one.

Public Member Functions

score

```
virtual double score (const ccCaliperModel& model,
  ccCaliperResult& candidate, ccCaliperData& res) const = 0;
```

Returns a score for the given edge in the input image. The method for computing the score depends on the scoring method type. The returned score is normalized to a value between 0.0 and 1.0.

Parameters

model The ideal edge, as defined by the user.

candidate The edge encountered in the input image

res The result for this edge

Notes

If the implementation of this function returns -1, the score will not be considered as part of the overall score.

clone

```
virtual ccCaliperScore* clone () const = 0;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const = 0;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccCallback

```
#include <ch_cvl/callback.h>

class ccCallback : public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

ccCallback is an abstract class for defining callback functions. Some CVL classes allow you to specify a function that will be called at a specific time. For example, the **ccAcqFifo** class, allows you to specify a function when it is safe to move the object in the camera's field of view.

To define a callback function, you derive a class from **ccCallback** and define **operator()**, the function call operator. Your derived class can define data members if necessary.

Constructors/Destructors

The default constructor and destructors do nothing. If you derive classes from this class, your constructor and destructors are responsible for allocating and releasing memory used by data members.

Operators

operator()

```
virtual void operator()() = 0;
```

Your derived class must define this operator to perform the actions that your callback function takes.

Typedefs

ccCallbackPtrh

```
typedef ccPtrHandle<ccCallback> ccCallbackPtrh;
```

ccCallbackPtrh_const

```
typedef ccPtrHandle_const<ccCallback>
  ccCallbackPtrh_const;
```

ccCallback

ccCallback1

```
#include <ch_cvl/callback.h>

template <class P> class ccCallback1 : public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	No

ccCallback1 is an template class for defining one-argument callback functions. See **ccCallback** for additional information.

P — The type of the lone argument passed to the function call operator.

To define a callback function, derive a class from **ccCallback1** and define **operator()**, the function call operator. Your derived class can define data members if necessary.

Constructors/Destructors

The default constructor and destructors do nothing. If you derive classes from this class, your constructor and destructors are responsible for allocating and releasing memory used by data members.

Operators

operator()

```
virtual void operator()(P arg1) = 0;
```

Your derived class must define this operator to perform the actions that your callback function takes.

Parameters

arg1 — The argument of type *P* passed to the function call operator.

Typedefs

CB1

```
typedef ccCallback1<const void*> CB1;
```

ccCallback1Ptrh

```
typedef ccPtrHandle<CB1> ccCallback1Ptrh;
```

ccCallback1Ptrh_const

```
typedef ccPtrHandle_const<CB1> ccCallback1Ptrh_const;
```

ccCallback2

```
#include <ch_cvl/callback.h>

template <class P, class Q> class ccCallback2 :
  public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	No

ccCallback2 is an template class for defining two-argument callback functions. See **ccCallback** for additional information.

P — The type of the first argument passed to the function call operator.

Q — The type of the second argument passed to the function call operator.

To define a callback function, derive a class from **ccCallback2** and define **operator()**, the function call operator. Your derived class can define data members if necessary.

Constructors/Destructors

The default constructor and destructors do nothing. If you derive classes from this class, your constructor and destructors are responsible for allocating and releasing memory used by data members.

Operators

operator()

```
virtual void operator()(P arg1,Q arg2) = 0;
```

Your derived class must define this operator to perform the actions that your callback function takes.

Parameters

arg1 — The argument of type *P* passed to the function call operator.

arg2 — The argument of type *Q* passed to function call operator.

ccCallback2

ccCameraPort

```
#include <ch_cvl/cogdevc.h>

class ccCameraPort;
```

Class Properties

Copyable	Not intended
Derivable	Yes
Archiveable	No

ccCameraPort is a class used to represent a camera port. This is useful for systems where the camera and camera type can be detected. For these systems the camera location property (**ccCameraLocationProp**) can be set via a reference to the found camera port.

Constructors/Destructors

ccCameraPort

```
protected: ccCameraPort(ccFrameGrabber& fg);

virtual ~ccCameraPort() {};
```

- `protected: ccCameraPort(ccFrameGrabber& fg);`

  Protected constructor, for Cognex use only.

  Parameters

  fg Frame grabber type.

- `virtual ~ccCameraPort();`

  Destructor.

Public Member Functions

frameGrabberOwner

```
ccFrameGrabber& frameGrabberOwner() const;
```

Returns a reference to the frame grabber that owns this camera port.

supportedFormat

```
bool supportedFormat(const ccVideoFormat&) const;

virtual bool supportedFormat(const cc_VideoFormat&)
   const = 0;
```

Parameters

ccVideoFormat A video format.

Returns true if the given video format is supported for this camera; otherwise returns false.

supportedFormats

```
virtual const ccVideoFormatList& supportedFormats()
   const = 0;
```

Returns a list of video formats supported by this camera.

newAcqFifo

```
ccStdGreyAcqFifo* newAcqFifo(
     const ccStdGreyVideoFormat& vf) const;
```

Parameters

vf A supported video format for this camera.

Returns a newly created acquisition FIFO for this camera using the given video format.

Notes

The **ccCameraLocationProp** is automatically set to use this camera.

Throws

ccCameraPort::BadParams()

If the given video format is not compatible with this camera.

Typedefs

ccVideoFormatList

```
typedef cmStd vector<const ccVideoFormat*>
   ccVideoFormatList;
```

ccCameraPortProp

```
#include <ch_cvl/prop.h>

class ccCameraPortProp;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes the camera port property of an acquisition FIFO queue.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccCameraPortProp

```
ccCameraPortProp();

explicit ccCameraPortProp(c_Int32 port);
```

- `ccCameraPortProp();`

  Creates a new camera port property not associated with any FIFO. The camera port is set to *ccCameraPort::defaultCameraPort*.

- `explicit ccCameraPortProp(c_Int32 port);`

  Creates a new camera port property not associated with any FIFO. The camera port is set to *port*.

  Parameters

port	The camera port to use.

Throws

ccCameraPortProp::BadParams
port is not in the range 0 to **numCameraPort()** - 1.

Public Member Functions

cameraPort

```
void cameraPort(c_Int32 port);
c_Int32 cameraPort();
```

- `void cameraPort(c_Int32 port);`

Sets the camera port that the acquisition FIFO this property is associated with should use.

Parameters

port The camera port to use.

Throws

ccCameraPortProp::BadParams
port is not in the range 0 to **numCameraPort()** - 1.

- `c_Int32 cameraPort() const;`

Returns the camera port that the acquisition FIFO associated with this property uses.

numCameraPort

```
c_Int32 numCameraPort() const;
```

Returns the number of camera ports that the underlying frame grabber supports.

Throws

ccCameraPortProp::NoFrameGrabber
This property is not associated with a **ccAcqFifo** object (and thus has no underlying frame grabber).

Constants

defaultCameraPort

```
static const c_Int32 defaultCameraPort;
```

The default camera port: 0.

ccCDBFile

```
#include <ch_cvl/cdb.h>

class ccCDBFile;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

The **ccCDBFile** lets you read from, write to, and navigate through a CDB file.

Constructors/Destructors

ccCDBFile

```
ccCDBFile();

ccCDBFile(const ccCvlString &name);

ccCDBFile(const ccCvlString &name, c_Int32 mode);

ccCDBFile(cmStd iostream* str,
  c_Int32 mode = cmStd ios::in | cmStd ios::out,
  bool becomeOwner = true);

~ccCDBFile();
```

- `ccCDBFile();`

  Constructs an unnamed and unbound **ccCDBFile** object.

- `ccCDBFile(const ccCvlString &name);`

  Constructs a **ccCDBFile** for read-only access to a CDB file.

  Parameters

  name The name of the CDB file.

  Throws

  ccCDBFile::CanNotOpenFile
  Cannot open the specified file. Verify that it exists and that it is not already open.

ccCDBFile::BadFileFormat
The specified file is not a valid CDB file.

- `ccCDBFile(const ccCvlString &name, c_Int32 mode);`

Constructs a **ccCDBFile** using the specified access mode on the specified CDB file.

Parameters

name The name of the CDB file.

mode The mode with which to open the file. *mode* must be formed by ORing together one or more of the following values:

cmStd ios::in
cmStd ios::out
cmStd ios::trunc

If you specify *cmStd ios::in*, the file is opened for read-only access. If you specify *cmStd ios::out*, the file is opened for writing. If you specify *cmStd ios::trunc* and *cmStd ios::out*, the file's contents are deleted as it is opened.

Throws

ccCDBFile::CanNotOpenFile
Cannot open the specified file. Verify that it exists, that it is not already open, and that you have write permission.

ccCDBFile::BadFileFormat
The specified file is not a valid CDB file.

ccCDBFile::InvalidArgument
mode is not a valid combination of *cmStd ios::in*, *cmStd ios::out*, and *cmStd ios::trunc*.

- ```
  ccCDBFile(cmStd iostream* str,
    c_Int32 mode = cmStd ios::in | cmStd ios::out,
    bool becomeOwner = true);
  ```

Constructs a **ccCDBFile** object that uses the specified **iostream** for input and output operations. If the specified **iostream** does not derive from **cmStd fstream**, calling **open()**, **filename()**, or **isOpen()** on this **ccCDBFile** throws *ccCDBFile::FileNotBound*.

Parameters

str The **iostream** to use for I/O.

mode The mode with which *str* was opened.
  ```

becomeOwner If true, then *str* will be deleted by **ccCDBFile**'s destructor when it is called. If false, then *str* will not be deleted.

Throws

ccCDBFile::InvalidArgument
str is null.

Notes

Calling **close()** flushes the CDB contents to *str*. **close()** is also called by **ccCDBFile**'s destructor.

- `~ccCDBFile();`

If the file associated with this **ccCDBFile** is open, calling the destructor closes the file.

Throws

ccCDBFile::WriteError
There was a file error trying to close the file.

Public Member Functions

open

```
void open(const ccCvlString &name,
  c_Int32 mode = cmStd ios::in);
```

Opens or creates a file with the specified name and mode. File pointer is positioned at the beginning of the first record; a subsequent call to **loadRecord()** loads the first record in the file.

Parameters

name The name of the CDB file.

mode The mode with which to open the file. *mode* must be formed by ORing together one or more of the following values:

cmStd ios::in
cmStd ios::out
cmStd ios::trunc

If you specify *cmStd ios::in*, the file is opened for read-only access. If you specify *cmStd ios::out*, the file is opened for writing. If you specify *cmStd ios::trunc* and *cmStd ios::out*, the file's contents are deleted as it is opened.

Throws

ccCDBFile::CanNotOpenFile
Cannot open the specified file. Verify that it exists, that it is not already open, and that you have write permission.

ccCDBFile::BadFileFormat
The specified file is not a valid CDB file.

ccCDBFile::InvalidArgument
mode is not a valid combination of *cmStd ios::in*, *cmStd ios::out*, and *cmStd ios::trunc*.

close `void close();`

Flushes the contents of the stream and closes the file. This **ccCDBFile** becomes unbound.

Throws

ccCDBFile::WriteError
There was a file error trying to close the file.

ccCDBFile::FileNotOpen
The file was not open.

filename `ccCvlString filename() const;`

Returns the name of the file associated with this **ccCDBFile**.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

isOpen `bool isOpen() const;`

Returns true if an open file is associated with this **ccCDBFile**.

loadRecord `void loadRecord(ccCDBRecord &rec);`

Reads the record at the current file pointer position in this **ccCDBFile**'s file into the supplied **ccCDBRecord**.

Parameters

rec A **ccCDBRecord** into which the record is read.

Throws

ccCDBFile::BadFileFormat
The specified file is not a valid CDB file.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

ccCDBFile::BadRecordFormat
The current record is an image record that has a bit depth other than 8.

ccCDBFile::RecordNotFound
The current record is END_OF_FILE.

storeRecord `void storeRecord(const ccCDBRecord &rec);`

Writes the supplied **ccCDBRecord** to the current file pointer position in this **ccCDBFile**'s open file.

Parameters
rec The **ccCDBRecord** to write.

Throws
ccCDBFile::RecordNotBound
The supplied **ccCDBRecord** does not have a type.

ccCDBFile::WriteError
There was a file error trying to write to the file.

ccCDBFile::RecordSizeMismatch
The size of the record being overwritten does not match the size of the supplied **ccCDBRecord**.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

appendRecord `void appendRecord(const ccCDBRecord &rec);`

Appends the supplied **ccCDBRecord** to the end of this **ccCDBFile**'s open file.

Parameters
rec The **ccCDBRecord** to write.

Throws
ccCDBFile::RecordNotBound
The supplied **ccCDBRecord** does not have a type.

ccCDBFile::WriteError
There was a file error trying to write to the file.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

deleteRecord `void deleteRecord();`

Deletes the last record from the end of this **ccCDBFile**'s open file.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Notes

This function works by changing the type of the last record in the file to *DELETE_TYPE*.

Records of type *INDEX_TYPE*, *EOF_TYPE* or *DELETE_TYPE* are not affected by this function.

seekToMatchingRecord

```
void seekToMatchingRecord(c_Int32 rec_count,
  const ccCDBRecord &rec,
  c_Int32 mode = ccCDBRecord::SM_TYPE,
  cmStd ios::seekdir origin = cmStd ios::beg);
```

Advances the file pointer in this **ccCDBFile**'s open file by the specified number of records that match the supplied template **ccCDBRecord**.

Parameters

rec_count — The number of records to advance. Specify a negative value for *rec_count* to move backwards through the file.

rec — The template **ccCDBRecord**. Only records that match the template record are counted as the file pointer is advanced.

mode — The search mode. *mode* lets you specify what constitutes a match between a record in this **ccCDBFile**'s open file and the supplied template record. *mode* must be formed by ORing together one or more of the following values:

ccCDBRecord::SM_TYPE
ccCDBRecord::SM_STYPE
ccCDBRecord::SM_DSIZE
ccCDBRecord::SM_SEQ
ccCDBRecord::SM_COMMENT

origin — Specify *cmStd ios::beg* to search from the start of the file, *cmStd ios::cur* to search from the current file pointer location, and *cmStd ios::end* to search from the end of the file.

Throws

ccCDBFile::InvalidArgument
rec_count is 0, *mode* or *origin* is invalid.

ccCDBFile::RecordNotFound
No matching record was found.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Notes

The file pointer is set at the beginning of the found record. Calling **loadRecord()** loads the found record.

findNextRecord

```
void findNextRecord(const ccCDBRecord &rec,
 c_Int32 mode = ccCDBRecord::SM_TYPE);
```

Advances the file pointer in this **ccCDBFile**'s open file to the next record that matches the supplied template **ccCDBRecord**.

Parameters

rec — The template **ccCDBRecord**. The file pointer is advanced to the next record that matches the template record.

mode — The search mode. *mode* lets you specify what constitutes a match between a record in this **ccCDBFile**'s open file and the supplied template record. *mode* must be formed by ORing together one or more of the following values:

ccCDBRecord::SM_TYPE
ccCDBRecord::SM_STYPE
ccCDBRecord::SM_DSIZE
ccCDBRecord::SM_SEQ
ccCDBRecord::SM_COMMENT

Throws

ccCDBFile::InvalidArgument
mode is invalid.

ccCDBFile::RecordNotFound
No matching record was found.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Notes

The file pointer is set at the beginning of the found record. Calling **loadRecord()** will load the found record.

findPrevRecord

```
void findPrevRecord(const ccCDBRecord &rec,
  c_Int32 mode = ccCDBRecord::SM_TYPE);
```

Moves the file pointer in this **ccCDBFile**'s open file to the first preceding record that matches the supplied template **ccCDBRecord**.

Parameters

rec — The template **ccCDBRecord**. The file pointer is moved to the first preceding record that matches the template record.

mode — The search mode. *mode* lets you specify what constitutes a match between a record in this **ccCDBFile**'s open file and the supplied template record. *mode* must be formed by ORing together one or more of the following values:

ccCDBRecord::SM_TYPE
ccCDBRecord::SM_STYPE
ccCDBRecord::SM_DSIZE
ccCDBRecord::SM_SEQ
ccCDBRecord::SM_COMMENT

Throws

ccCDBFile::InvalidArgument
mode is invalid.

ccCDBFile::RecordNotFound
No matching record was found.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Notes

The file pointer is set at the beginning of the found record. Calling **loadRecord()** will load the found record.

goToRecord

```
void goToRecord(c_Int32 recnum,
  cmStd ios::seekdir origin = cmStd ios::beg);
```

Moves the file pointer in this **ccCDBFile**'s open file to the specified record.

Parameters

recnum — The record number to advance to.

origin — Specify *cmStd ios::beg* to advance from the start of the file, *cmStd ios::cur* to advance from the current file pointer location, and *cmStd ios::end* to move backward from the end of the file.

Throws

ccCDBFile::InvalidArgument
origin is invalid.

ccCDBFile::RecordNotFound
The specified record number is not present in this file.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Notes

The number of the first record in the file is 0.

If *recnum* = **numRecords()**, the file pointer is set to immediately after the last record (END_OF_FILE).

nextRecord

```
void nextRecord();
```

Advances to the next record in this **ccCDBFile**'s open file.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

ccCDBFile::RecordNotFound
The specified record number is not present in this file.

prevRecord

```
void prevRecord();
```

Moves to the previous record in this **ccCDBFile**'s open file.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

ccCDBFile::RecordNotFound
The specified record number is not present in this file.

firstRecord

```
void firstRecord();
```

Sets the file pointer to point to the first record in this **ccCDBFile**'s open file.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

lastRecord `void lastRecord();`

Sets the file pointer to point to the last record in this **ccCDBFile**'s open file.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

currentRecord `c_Int32 currentRecord() const;`

Returns the number of the current record. If the file pointer is immediately after the last record, this function returns *ccCDBFile::END_OF_FILE*.

Throws

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

numRecords

```
c_Int32 numRecords(const ccCDBRecord &rec,
  c_Int32 mode = ccCDBRecord::SM_TYPE) const;

c_Int32 numRecords() const;
```

- ```
  c_Int32 numRecords(const ccCDBRecord &rec,
    c_Int32 mode = ccCDBRecord::SM_TYPE) const;
  ```

  Returns the number of records in this **ccCDBFile**'s open file that match the supplied template **ccCDBRecord**.

  Parameters

  rec — The template **ccCDBRecord**. Only records that match the template record are counted.

  mode — The search mode. *mode* lets you specify what constitutes a match between a record in this **ccCDBFile**'s open file and the supplied template record. *mode* must be formed by ORing together one or more of the following values:

  ccCDBRecord::SM_TYPE
  ccCDBRecord::SM_STYPE
  ccCDBRecord::SM_DSIZE
  ccCDBRecord::SM_SEQ
  ccCDBRecord::SM_COMMENT

  Throws

  ccCDBFile::InvalidArgument
  mode is invalid.
  ```

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

- `c_Int32 numRecords() const;`

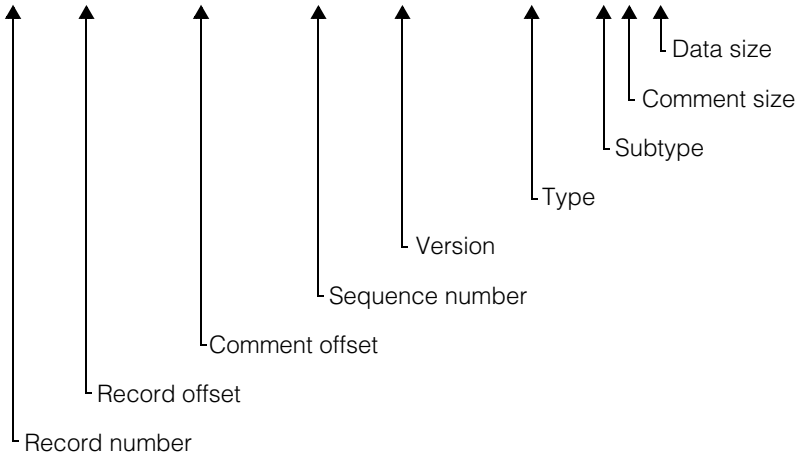
  Returns the total number of records in this **ccCDBFile**'s open file.

  Throws
  ccCDBFile::FileNotBound
  There is no open file associated with this **ccCDBFile**.

dumpIndex `ccCvlOStream& dumpIndex(cmStd ccCvlOStream& out) const;`

Prints the index from this **ccCDBFile**'s open file using the same format as the Cognex PVE function **cdb_dump_index()**. An example of this format is shown below:

```
0000      0      36: 00000 19910404 IMAG 1 0  30798
0001  30834   30870: 00001 19910404 RSLT 1 0     34
0002  30904   30940: 00002 19910404 IMAG 0 0 230702
```



Parameters
out The stream to which to print the index.

getRecordHeader

`ccCDBRecordHeader getRecordHeader(c_Int32 recnum) const;`

Returns a **ccCDBRecordHeader** containing information about the specified record in this **ccCDBFile**'s open file.

Parameters

recnum The record about which to obtain information.

Throws

ccCDBFile::RecordNotFound
The specified record number is not present in this file.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Deprecated Members

findRecord

```
void findRecord(c_Int32 rec_count, const ccCDBRecord &rec,
  c_Int32 mode = ccCDBRecord::SM_TYPE,
  cmStd ios::seekdir origin = cmStd ios::beg);
```

Notes

This function has been deprecated. Use **seekToMatchingRecord()** instead.

Advances the file pointer in this **ccCDBFile**'s open file by the specified number of records that match the supplied template **ccCDBRecord**. If the first record in this **ccCDBFile**'s open file matches the supplied template **ccCDBRecord**, the pointer is advanced by the specified number of records plus one.

Parameters

rec_count The number of records to advance. Specify a negative value for *rec_count* to move backwards through the file.

rec The template **ccCDBRecord**. Only records that match the template record are counted as the file pointer is advanced.

mode The search mode. *mode* lets you specify what constitutes a match between a record in this **ccCDBFile**'s open file and the supplied template record. *mode* must be formed by ORing together one or more of the following values:

ccCDBRecord::SM_TYPE
ccCDBRecord::SM_STYPE
ccCDBRecord::SM_DSIZE
ccCDBRecord::SM_SEQ
ccCDBRecord::SM_COMMENT

origin Specify *cmStd ios::beg* to search from the start of the file, *cmStd ios::cur* to search from the current file pointer location, and *cmStd ios::end* to search from the end of the file.

Throws

ccCDBFile::InvalidArgument
rec_count is 0, *mode* or *origin* is invalid.

ccCDBFile::RecordNotFound
No matching record was found.

ccCDBFile::FileNotBound
There is no open file associated with this **ccCDBFile**.

Notes

The file pointer is set at the beginning of the found record. Calling **loadRecord()** loads the found record.

ccCDBRecord

```
#include <ch_cvl/cdb.h>

class ccCDBRecord;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **ccCDBRecord** class implements a single CDB record. The **ccCDBFile** class provides support for reading **ccCDBRecord**s from and writing **ccCDBRecord**s to a CDB file.

Constructors/Destructors

ccCDBRecord

```
ccCDBRecord();

ccCDBRecord(c_Int32 type, c_Int32 subtype = 0,
  size_t dataSize = 0, const void *data = 0,
  const ccCvlString comment = cmT(""),
  c_Int32 dataVersion = 0, c_Int32 seqNum = 1);
```

- `ccCDBRecord();`

  Constructs an unbound **ccCDBRecord**.

- ```
  ccCDBRecord(c_Int32 type, c_Int32 subtype = 0,
    size_t dataSize = 0, const void *data = 0,
    const ccCvlString comment = cmT(""),
    c_Int32 dataVersion = 0, c_Int32 seqNum = 1);
  ```

  Constructs and initializes a **ccCDBRecord**.

  Parameters

  type — The record type. *type* must be one of the following values:

  ccCDBRecord::COMMAND_TYPE
  ccCDBRecord::RESULT_TYPE
  ccCDBRecord::IMAGE_TYPE
  ccCDBRecord::PERF_TYPE
  ccCDBRecord::ACQUIRE_TYPE
  ```

ccCDBRecord::MODEL_TYPE
ccCDBRecord::HEADER_TYPE
ccCDBRecord::RESPONSE_TYPE
ccCDBRecord::INDEX_TYPE
ccCDBRecord::EOF_TYPE
ccCDBRecord::TDB_TYPE
ccCDBRecord::IMAGE_ROI_TYPE
ccCDBRecord::DELETE_TYPE

subtype	The record subtype. This field is not used in CVL.
dataSize	The number of bytes of data to include in this record.
data	A pointer to the data for this record. *data* should point to at least *dataSize* bytes of storage.
comment	Comment data to store as part of this record. *comment* must point to a null-terminated string.
dataVersion	A version number to write as part of this record. You can specify any value for *dataVersion*.
seqNum	A sequence number. You can specify any value for *seqNum*.

Enumerations

Record Type This enumeration defines the CDB record types.

Value	Meaning
COMMAND_TYPE	Cognex internal use.
RESULT_TYPE	Cognex internal use.
IMAGE_TYPE	Image data
PERF_TYPE	Cognex internal use.
ACQUIRE_TYPE	Image data
MODEL_TYPE	Cognex internal use.
HEADER_TYPE	Cognex internal use.
RESPONSE_TYPE	Cognex internal use.
INDEX_TYPE	Cognex internal use.
EOF_TYPE	End of file marker record

Value	Meaning
TDB_TYPE	Cognex internal use.
IMAGE_ROI_TYPE	Cognex internal use.
DELETE_TYPE	This record marked for deletion

Notes

Unlike most CVL enumerations, there is no type associated with this enumeration.

Search Mode

This enumeration defines the types of search modes supported when searching for a **ccCDBRecord** within a CDB file.

Value	Meaning
SM_SEQ	Match the sequence number.
SM_TYPE	Match the record type.
SM_STYPE	Match the record subtype.
SM_DSIZE	Match the data size.
SM_COMMENT	Match the comment data.

Notes

Unlike most CVL enumerations, there is no type associated with this enumeration.

Sub-Type

This enumeration defines the types of images that may be contained in an image record.

Value	Meaning
SUBTYPE_IMG_GRAY6	6-bit greyscale image.
SUBTYPE_IMG_GRAY8	8-bit greyscale image.
SUBTYPE_IMG_COLOR24	24-bit color image.
SUBTYPE_IMG_GRAY16	16-bit greyscale image.

Notes

Not all 8-bit images have a sub-type of SUBTYPE_IMG_GRAY8; to test if an image is an 8-bit image, the sub-type must be neither SUBTYPE_IMG_COLOR24 nor SUBTYPE_IMG_GRAY16.

Unlike most CVL enumerations, there is no type associated with this enumeration.

Public Member Functions

fillRecord

```
void fillRecord(c_Int32 type, c_Int32 subtype = 0,
  size_t dataSize = 0, const void *data = 0,
  const ccCvlString comment = cmT(""),
  c_Int32 dataVersion = 0, c_Int32 seqNum = 1);
```

Initializes an unbound **ccCDBRecord**.

Parameters

type The record type. *type* must be one of the following values:

ccCDBRecord::COMMAND_TYPE
ccCDBRecord::RESULT_TYPE
ccCDBRecord::IMAGE_TYPE
ccCDBRecord::PERF_TYPE
ccCDBRecord::ACQUIRE_TYPE
ccCDBRecord::MODEL_TYPE
ccCDBRecord::HEADER_TYPE
ccCDBRecord::RESPONSE_TYPE
ccCDBRecord::INDEX_TYPE
ccCDBRecord::EOF_TYPE
ccCDBRecord::TDB_TYPE
ccCDBRecord::IMAGE_ROI_TYPE
ccCDBRecord::DELETE_TYPE

subtype The record subtype. This meaning of the *subtype* depends on the record type.

dataSize The number of bytes of data to include in this record.

data A pointer to the data for this record. *data* should point to at least *dataSize* bytes of storage.

comment Comment data to store as part of this record. *comment* must point to a null-terminated string.

dataVersion A version number to write as part of this record. You can specify any value for *dataVersion*.

seqNum A sequence number. You can specify any value for *seqNum*.

Throws

ccCDBRecord::WrongType
type is not one of the enumerated record types.

freeRecord

```
void freeRecord();
```

Unbinds this **ccCDBRecord**.

type

```
c_Int32 type();
```

Returns the type of this **ccCDBRecord**. The value returned by this function is one of the following:

ccCDBRecord::COMMAND_TYPE
ccCDBRecord::RESULT_TYPE
ccCDBRecord::IMAGE_TYPE
ccCDBRecord::PERF_TYPE
ccCDBRecord::ACQUIRE_TYPE
ccCDBRecord::MODEL_TYPE
ccCDBRecord::HEADER_TYPE
ccCDBRecord::RESPONSE_TYPE
ccCDBRecord::INDEX_TYPE
ccCDBRecord::EOF_TYPE
ccCDBRecord::TDB_TYPE
ccCDBRecord::IMAGE_ROI_TYPE
ccCDBRecord::DELETE_TYPE

Throws

ccCDBRecord::RecordNotBound
This record is unbound.

version

```
c_Int32 version() const;
```

Returns the version of the CDB software used to create this **ccCDBRecord**.

Throws

ccCDBRecord::RecordNotBound
This record is unbound.

dataVersion

```
c_Int32 dataVersion() const;
void dataVersion(c_Int32 vers);
```

- `c_Int32 dataVersion() const;`

Returns the data version of this **ccCDBRecord**.

- `void dataVersion(c_Int32 vers);`

  Sets the data version of this **ccCDBRecord**. You can set the data version to any value.

  Parameters
  vers The data version to set.

  Throws
  ccCDBRecord::RecordNotBound
  This record is unbound.

sequenceNumber

```
c_Int32 sequenceNumber() const;

void sequenceNumber(c_Int32 num);
```

- `c_Int32 sequenceNumber() const;`

  Returns the sequence number of this **ccCDBRecord**.

- `void sequenceNumber(c_Int32 vers);`

  Sets the sequence number of this **ccCDBRecord**. You can set the sequence number to any value.

  Parameters
  vers The sequence number to set.

  Throws
  ccCDBRecord::RecordNotBound
  This record is unbound.

subtype

```
c_Int32 subtype() const;

void subtype(c_Int32 num);
```

- `c_Int32 subtype() const;`

  Returns the subtype of this **ccCDBRecord**. The value returned by this function is one of the following:

  ccCDBRecord::SUBTYPE_IMG_GRAY6
  ccCDBRecord::SUBTYPE_IMG_GRAY8
  ccCDBRecord::SUBTYPE_IMG_COLOR24
  ccCDBRecord::SUBTYPE_IMG_GRAY16

- `void sequenceNumber(c_Int32 type);`

  Sets the subtype of this **ccCDBRecord**.

  Parameters

  type The subtype to set.

  Throws

  ccCDBRecord::RecordNotBound
  This record is unbound.

  Notes

  The subtype is used internally; you should not attempt to set the subtype of a record yourself.

dataSize

`size_t dataSize() const;`

Returns the number of bytes of data in this record. If the record type is *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*, then this function returns the number of bytes in the image.

Throws

ccCDBRecord::RecordNotBound
This record is unbound.

image

`ccPelBuffer_const<c_UInt8> image() const;`

`void image(const ccPelBuffer_const<c_UInt8>& img);`

- `ccPelBuffer_const<c_UInt8> image() const;`

  Returns a **ccPelBuffer** representation of this **ccCDBRecord**'s image data. This **ccCDBRecord**'s type must be *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*.

  Throws

  ccCDBRecord::WrongSubType
  This record has a subtype of *ccCDBRecord::SUBTYPE_IMG_GRAY16*.

- `void image(const ccPelBuffer_const<c_UInt8>& img);`

  Sets the contents of this **ccCDBRecord** to be the supplied **ccPelBuffer** image data. This **ccCDBRecord**'s type must be *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*

Parameters

img The image to store.

Throws

ccCDBRecord::RecordNotBound
This record is unbound.

ccCDBRecord::WrongType
This **ccCDBRecord**'s type is not *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*.

ccCDBRecord::WrongSubType
This **ccCDBRecord**'s has a subtype of *ccCDBRecord::SUBTYPE_IMG_GRAY16*.

Notes

The **ccPelBuffer**'s offset and client coordinate system are not saved in the CDB file. When you load a **ccPelBuffer**, the offset is set to (0,0) and the client coordinate system is the same as the image coordinate system.

---

image16

```
ccPelBuffer_const<c_UInt16> image16() const;

void image16(const ccPelBuffer_const<c_UInt16>&,
   ceImageFormat = ceImageFormat_Unknown);
```

- `ccPelBuffer_const<c_UInt16> image16() const;`

Returns a **ccPelBuffer** representation of this **ccCDBRecord**'s 16-bit image data. This **ccCDBRecord**'s type must be *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*.

Throws

ccCDBRecord::WrongSubType
This record has a subtype of *ccCDBRecord::SUBTYPE_IMG_GRAY16*.

- `void image16(const ccPelBuffer_const<c_UInt16>&, ceImageFormat = ceImageFormat_Unknown);`

Sets the contents of this **ccCDBRecord** to be the supplied **ccPelBuffer** image data. This **ccCDBRecord**'s type must be *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*

Parameters

img The image to store.

Throws

ccCDBRecord::RecordNotBound
This record is unbound.

ccCDBRecord::WrongType
This **ccCDBRecord**'s type is not *ccCDBRecord::IMAGE_TYPE* or *ccCDBRecord::ACQUIRE_TYPE*.

ccCDBRecord::WrongSubType
This **ccCDBRecord**'s has a subtype of *ccCDBRecord::SUBTYPE_IMG_GRAY16*.

Notes

The **ccPelBuffer**'s offset and client coordinate system are not saved in the CDB file. When you load a **ccPelBuffer**, the offset is set to (0,0) and the client coordinate system is the same as the image coordinate system.

comment

```
ccCvlString comment() const;
void comment(const ccCvlString &str);
```

- `ccCvlString comment() const;`

Returns the comment string for this **ccCDBRecord**.

- `void comment(const ccCvlString &str);`

Sets the comment string for this **ccCDBRecord**. You can store any value in this string. You can use the **ccCDBFile::findRecord()** function to search a CDB file for **ccCDBRecord**s with a particular comment string.

Parameters

str The comment string. *str* must be a null-terminated string.

Throws

ccCDBRecord::RecordNotBound
This record is unbound.

Static Functions

typeString

```
static ccCvlString typeString(int subtype);
```

Returns a 4-character string description of the type.

subtypeString `static ccCvlString subtypeString(int subtype);`

Returns a 4-character string description of the subtype.

ccCharCode

```
#include <ch_cvl/charcode.h>

class ccCharCode;
```

A name space for constants and utility functions for character codes.

Public Member Functions

isReserved16 `static bool isReserved16(c_Char32 c);`

Returns whether *c* is within the Cognex-reserved 16-bit range.

Parameters

c The character code.

isReserved `static bool isReserved(c_Char32 c);`

Return whether *c* is within a Cognex-reserved range.

Parameters

c The character code.

isSpace `static bool isSpace(c_Char32 characterCode);`

Returns whether *characterCode* is any whitespace character according to the UTF-32 code table.

Parameters

characterCode The character code.

Notes

This returns false for any Cognex-reserved character code.

isRepresentableAsChar
`static bool isRepresentableAsChar(c_Char32 c);`

Returns whether *c* is representable as type char. A character code is representable only if it keeps the same value when cast to the given type.

Parameters

c The character code.

isRepresentableAsChar16

```
static bool isRepresentableAsChar16(c_Char32 c);
```

Returns whether *c* is representable as type c_Char16. A character code is representable only if it keeps the same value when cast to the given type.

Parameters

c The character code.

isRepresentableAsWChar

```
static bool isRepresentableAsWChar(c_Char32 c);
```

Returns whether *c* is representable as type wchar_t. A character code is representable only if it keeps the same value when cast to the given type.

Parameters

c The character code.

isRepresentableAsTChar

```
static bool isRepresentableAsTChar(c_Char32 c);
```

Returns whether *c* is representable as type TCHAR. A character code is representable only if it keeps the same value when cast to the given type.

Parameters

c The character code.

getAsChar `static char getAsChar(c_Char32 c);`

Convert *c* to the type char if possible.

Parameters

c The character code.

getAsChar16 `static c_Char16 getAsChar16(c_Char32 c);`

Convert *c* to the type c_Char16 if possible.

Parameters

c The character code.

getAsWChar `static wchar_t getAsWChar(c_Char32 c);`

Convert *c* to the type wchar_t if possible.

Parameters

c The character code.

getAsTChar

```
static TCHAR getAsTChar(c_Char32 c);
```

Convert *c* to the type TCHAR if possible.

Parameters

c The character code.

Throws

ccCharCodeDefs::NotRepresentable
c is not representable as the given type.

Constants

eReserved16Low

```
static const c_Char16 eReserved16Low = 0xF800;
```

eReserved16High

```
static const c_Char16 eReserved16High = 0xF8FF;
```

eReserved32Low

```
static const c_Char32 eReserved32Low = 0xFF000;
```

eReserved32High

```
static const c_Char32 eReserved32High = 0xFFFFD;
```

The bounds of the Cognex-reserved private use ranges.

eUnknown

```
static const c_Char16 eUnknown = 0xF800;
```

Specifies that the character code is unknown.

Notes

This value is in a Cognex-reserved private use range.

eAnyCharacter

```
static const c_Char16 eAnyCharacter = 0xF801;
```

This is used to specify that all non-reserved character codes are allowed.

Notes

This value is in a Cognex-reserved private use range.

eAnyNonspaceCharacter

```
static const c_Char16 eAnyNonspaceCharacter = 0xF802;
```

This is used to specify that all non-reserved character codes except the space character 0x20 are allowed.

Notes

This value is in a Cognex-reserved private use range.

ccCircle

```
#include <ch_cvl/shapes.h>

class ccCircle : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Not intended
Archiveable	Complex

This class describes a circle specified by a center and a radius.

Constructors/Destructors

ccCircle

```
ccCircle(const cc2Vect& c = cc2Vect(0.0, 0.0),
  double r = 0.0);
```

Constructs a circle with the specified center and radius.

Parameters

c The location of the center of the circle. The default location, if no value is supplied, is (0.0, 0.0).

r The radius of the circle. The default radius, if no value is supplied, is 0.0.

Operators

operator==

```
bool operator==(const ccCircle& other) const;
```

Returns true if this circle is equal to another circle.

Parameters

other The other circle.

operator!=

```
bool operator!=(const ccCircle& other) const;
```

Returns true if this circle is not equal to another circle.

Parameters

other The other circle.

Public Member Functions

center

```
const ccPoint& center() const;
```

Returns the center of the circle.

radius

```
double radius() const;
```

Returns the radius of the circle.

map2

```
ccEllipse2 map2(const cc2Xform& c) const;
```

Returns this circle mapped by the given transformation object.

Parameters

c	The transformation object.

degen

```
bool degen() const;
```

Returns true if this circle is degenerate (its radius is less than or equal to zero).

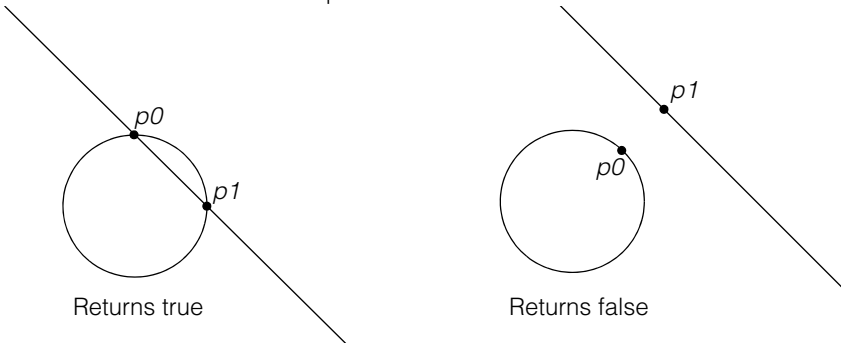
intersections

```
bool intersections(const ccLine &line, cc2Vect &point0,
  cc2Vect &point1) const;

bool intersections(const ccCircle &other, cc2Vect &point0,
  cc2Vect &point1) const;
```

- ```
  bool intersections(const ccLine &line, cc2Vect &point0,
    cc2Vect &point1) const;
  ```

  Returns true if this circle intersects the given line, and false if it does not. If the function returns true, *point0* and *point1* are returned as the intersection points. If the function returns false, *point0* and *point1* are returned as a pair of closest points between the circle and line. In this case, *point0* is the closest point on the circle, and *point1* is the closest point on the line. See the examples below:



Parameters

line	The given line to test for intersection with this circle.
point0	A returned point, dependent on whether the line and circle intersect. See description above.
point1	A returned point, dependent on whether the line and circle intersect. See description above.

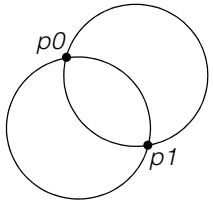
Notes

Regardless of the return value, if the computed *point0* and *point1* are very close, the circle and line are approximately tangent. The closeness criterion is application specific and likely depends on the circle's radius. If the circle and line are exactly tangent, the function returns true, and *point0* and *point1* are identical.

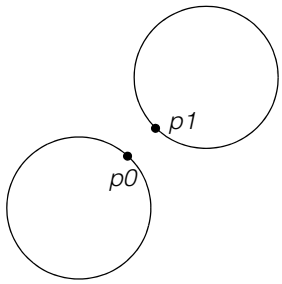
If the function returns true, the intersection points are returned in the order such that **line.offset(***point0***)** <= **line.offset(***point1***)**.
  ```

- `bool intersections(const ccCircle &other, cc2Vect &point0, cc2Vect &point1) const;`

Returns true if this circle intersects another circle, and false if it does not. If the function returns true, *point0* and *point1* are returned as the intersection points. If the function returns false, *point0* and *point1* are returned as a pair of closest points between the two circles. In this case, *point0* is the closest point on this circle, and *point1* is the closest point on the other circle.



Returns true



Returns false

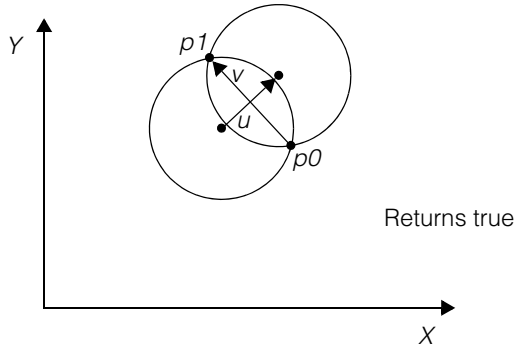
Parameters

other	The other circle to test for intersection with this circle.
point0	A returned point, dependent on whether the two circles intersect. See description above.
point1	A returned point, dependent on whether the two circles intersect. See description above.

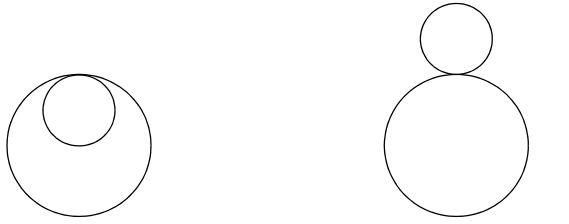
Notes

1. Regardless of the return value, if the computed *point0* and *point1* are very close, the circles are approximately tangent. The closeness criterion is application specific and likely depends on the circles' radii. If the two circles are exactly tangent, the function returns true, and *point0* and *point1* are identical.
2. Concentric circles have either an infinite number of intersection points or an infinite number of closest points, depending on whether their radii are equal or not. This function returns true when the radii are equal, and false if the radii are not equal. In either case, the returned points are arbitrarily chosen as the intersections of the circles with a ray passing through the circles' common center and aligned with the positive x-axis.

3. If the function returns true, the intersection points are returned in the order such that if u is the vector from this circle's center to the other circle's center, and v is the vector from *point0* to *point1*, then the cross product u x v is positive. See the following example:



4. The return value and returned points do not completely define all possible cases. For example, if the function returns true and the returned points are equal, the circles are tangent, however further analysis is required to determine if the tangency is exterior or interior. See the following examples.



Also, if the function returns false this circle may enclose the other circle, or vice versa, or neither may enclose the other. Here again, further analysis is required if these distinctions are significant.

clone `virtual ccShapePtrh clone() const;`

Returns a pointer to a copy of this circle.

isOpenContour `virtual bool isOpenContour() const;`

Returns true if this shape is an open contour. For circles, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion `virtual bool isRegion() const;`

Returns true if this shape is a region. For circles, this function always returns true, even circles that are degenerate. See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

For circles, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For circles, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

Thus function returns true for most circles. It returns false for circles that are degenerate.See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For circles, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible `virtual bool isReversible() const;`

For circles, this function always returns false. See **ccShape::reverse()** for more information.

boundingBox `virtual ccRect boundingBox() const;`

Returns the smallest rectangle that encloses this circle. See **ccShape::boundingBox()** for more information.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const;`

Returns the nearest point on the boundary of this circle to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

perimeter

```
virtual double perimeter() const;
```

Returns the perimeter of this circle.

See **ccShape::perimeter()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores circles for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

For circles, this function always returns true. Circles are always right-handed. See **ccShape::isRightHanded()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this circle mapped by *X*.

Parameters

X	The transformation object.

Notes

The returned shape is a **ccEllipse2**, unless *X* is the identity transform.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccEllipseArc2**s. See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect& p) const;
```

Returns true if the given point is within this circle.

Parameters

p The point.

See **ccShape::within()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

ccCircle

```
ccCircle(const ccEllipse&);
```

Use the constructor that specifies a center and radius instead of this constructor.

map

```
ccEllipse map(const cc2Xform& c);
```

Use **map2()** instead of this function.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

closestPoint

```
cc2Vect closestPoint(const cc2Vect &p) const;
```

Use **nearestPoint()** instead of this function.

ccCircleFitDefs

```
#include <ch_cvl/fit.h>

class ccCircleFitDefs;
```

A name space that holds enumerations and constants used with the Circle Fitting tool (**cfCircleFit()**).

Enumerations

fit_mode

```
enum fit_mode
```

This enumeration defines types of fitting supported by the Circle Fitting tool.

Value	Meaning
eLeastSquaresMeasureRadius	Fits a circle by minimizing the least squares error. Calculates the radius of the circle.
eLeastSquaresUseExpectedRadius	Fits a circle whose radius you specify by minimizing the least squares error.
kDefaultFitMode = eLeastSquaresMeasureRadius	

ccCircleFitDefs

ccCircleFitParams

```
#include <ch_cvl/fit.h>

class ccCircleFitParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds the Circle Fitting tool parameters. See **cfCircleFit()**.

Constructors/Destructors

ccCircleFitParams

```
ccCircleFitParams(
  enum ccCircleFitDefs::fit_mode fitMode =
                          ccCircleFitDefs::kDefaultFitMode,
  double radius=1.,
  c_UInt32 numIgnore=0,
  double threshold=HUGE_VAL);
```

Constructs a **ccCircleFitParams** object.

Parameters

fitMode — The type of fitting to perform. Must be one of the following:

ccCircleFitDefs:: eLeastSquaresMeasureRadius; the tool computes the circle radius.

ccCircleFitDefs::eLeastSquaresUseExpectedRadius; you provide the radius (see *radius* below). Can cause the tool to run faster but may produce a less accurate found circle position.

radius — The radius of the circle to fit. If *fitMode* is *ccCircleFitDefs::eLeastSquaresMeasureRadius*, then this value is ignored.

numIgnore — The number of outlying points to ignore. The tool always discards this number of points from the point set used to find the circle.

threshold — The maximum RMS error for a valid fit. If the fitted circle has an RMS error greater than *threshold*, it is not considered a valid fit.

Throws

ccCircleFitDefs::BadParams
radius is less than 0 or *threshold* is less than 0.

operator==

```
bool operator== (const ccCircleFitParams& that) const;
```

True if **this* equals *that*; false otherwise.

Two **ccCircleFitParams** objects are considered equal if and only if their fitMode, radius, numIgnore, and threshold values are each equal

Parameters

that The **ccCircleFitParams** object compared with this object.

Public Member Functions

fitMode

```
enum ccCircleFitDefs::fit_mode fitMode() const;
void fitMode(enum ccCircleFitDefs::fit_mode fitMode);
```

- `enum ccCircleFitDefs::fit_mode fitMode() const;`

  Returns the type of fitting this **ccCircleFitParams** is configured to perform. The function returns one of the following values:

  ccCircleFitDefs:: eLeastSquaresMeasureRadius
  ccCircleFitDefs::eLeastSquaresUseExpectedRadius

- `void fitMode(enum ccCircleFitDefs::fit_mode fitMode);`

  Sets the type of fitting this **ccCircleFitParams** is configured to perform.

  Parameters

  fitMode The type of fitting to perform. Supplying the radius of the circle requires less time; computing the radius of the circle is more flexible. *fitMode* must be one of the following values:

  ccCircleFitDefs:: eLeastSquaresMeasureRadius
  ccCircleFitDefs::eLeastSquaresUseExpectedRadius

radius

```
double radius() const;
void radius(double radius);
```

- `double radius() const;`

  Returns the expected radius of the circle being fit in client coordinate system units.

- `void radius(double radius);`

  Sets the expected radius of the circle being fit in client coordinate system units. This value is ignored if the tool is configured to compute the radius of the circle.

  Parameters

  radius The expected radius in client coordinate system units.

  Throws

  ccCircleFitDefs::BadParams
  radius is less than 0.

numIgnore

```
c_UInt32 numIgnore() const;
void numIgnore(c_UInt32 numIgnore);
```

- `c_UInt32 numIgnore() const;`

  Returns the number of outlying points ignored by this **ccCircleFitParams**.

- `void numIgnore(c_UInt32 numIgnore);`

  Sets the number of outlying points ignored by this **ccCircleFitParams**.

  Parameters

  numIgnore The number of points to ignore.

threshold

```
double threshold() const;
void threshold(double threshold);
```

- `double threshold() const;`

  Returns the maximum RMS error which a fit circle can receive and still be considered found.

- `void threshold(double threshold);`

  Sets the maximum RMS error which a fit circle can receive and still be considered found. If the RMS error is greater than or equal to the supplied threshold, the circle is not considered found.

  Parameters

  threshold The RMS error threshold. *threshold* can have any positive value.

  Throws

  ccCircleFitDefs::BadParams
  threshold is less than 0.

ccCircleFitResults

```
#include <ch_cvl/fit.h>

class ccCircleFitResults;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds the results from the Circle Fitting tool. See **cfCircleFit()**.

Constructors/Destructors

ccCircleFitResults

```
ccCircleFitResults();
```

Construct a **ccCircleFitResults** with no result.
found() = false, **error()** = 0, **time()** = 0.

Operators

operator==

```
bool operator== (const ccCircleFitResults& rhs) const;
```

Two **ccCircleFitResults** objects are considered equal if and only if the values returned by **found**, **circle**, and **error** are equal.

Public Member Functions

found

```
bool found() const;
```

Returns true if the RMS error of this circle is less than the threshold you supplied. Returns false otherwise.

reset

```
void reset();
```

Reset this **ccCircleFitResults**. Calling **found** returns false after calling this function.

ccCircleFitResults

circle

```
const ccCircle &circle() const;
```

Returns the fitted circle.

error

```
double error() const;
```

The measured RMS error between the data and the estimated circle.

The default value is 0.

runParams

```
const ccCircleFitParams &runParams() const;
```

Returns the **ccCircleFitParams** used to fit this circle. If **found** returns false, then this function's return value is undefined.

outliers

```
const cmStd vector<c_UInt32> &outliers() const;
```

Returns the indices of the points which were ignored. If **found** returns false, then this function's return value is undefined.

time

```
double time() const;
```

Returns the time in milliseconds required to compute this result. If **found** returns false, then this function's return value is undefined.

ccCircularLabeledProjectionModel

```
#include <ch_cvl/lablproj.h>

template <class M>
class ccCircularLabeledProjectionModel:
  ccLabeledProjectionModel<M>;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This is a templated class that contains a circular or radial projection model.

M — Template parameter specifying the value associated with each pixel in the model. CVL provides instantiations of **ccCircularLabeledProjectionModel** for the types **c_UInt8**, **c_UInt16**, and **c_UInt32**

This class is a specialization of the **ccLabeledProjectionModel** class. You specify parameters upon construction to create a circular or radial projection model. The constructor allocates an image and initializes it according to the parameters you supply.

Constructors/Destructors

ccCircularLabeledProjectionModel

```
ccCircularLabeledProjectionModel();
```

Returns an untrained **ccCircularLabeledProjectionModel**.

Public Member Functions

train

```
void train (c_Int32 radius,
  cc2Vect centerOffset=cc2Vect(0,0), double rMin=0,
  double rMax=0, c_Int32 numAnnuli=1, c_Int32 numSectors=1,
  enum ccLabeledProjection::BinOrder
  binOrder= ccLabeledProjection::kDefaultBinOrder,
  enum ccLabeledProjection::Orientation
```

```
orientation= ccLabeledProjection::kDefaultOrientation,
ccDegree startAngle=ccDegree(0.0),
ccDegree endAngle=ccDegree(360.0));
```

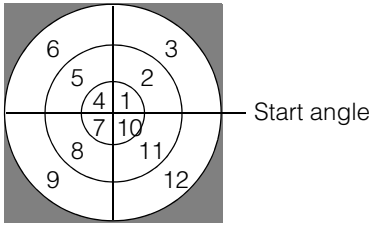
Trains this **ccCircularLabeledProjectionModel** according to the parameters you supply. This function will allocate and bind an image of the pixel type you specify with a height and width of *radius**2 + 1.

The constructor will initialize the pixel values in the image as follows:

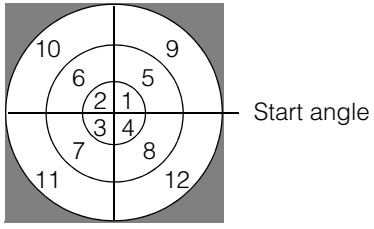
1. All pixels inside or on a circle of radius *rMin* or outside a circle of radius *rMax* are set to 0, indicating that they are *don't care* pixels. If *rMax* is 0, then no pixels outside of *rMin* are set to 0. Set *rMin* to zero to extend bins to the corners of the projection model.
2. If *orientation* is *ccLabeledProjection::eClockwise* (the default), pixels in the counter-clockwise range *endAngle* to *startAngle* are set to zero. The positive side of the client coordinate system x-axis is 0.
3. Divide up the sector between *startAngle* and *endAngle* into *numSectors* and then divide each sector between *rMin* and *rMax* into *numAnnuli*.
4. If *binOrder* is *ccLabeledProjection::eRadial*, then the bins are numbered starting with the innermost annulus of the first sector, then moving outward within the first sector, then from innermost to outermost in the next sector in the direction specified by *orientation*.

   If *binOrder* is *ccLabeledProjection::eAngular*, then the bins are numbered starting with the innermost annulus of the first sector in the direction specified by *orientation* from *startAngle*, then moving to the innermost annulus of the second sector, continuing until the entire inner annulus was filled, then moving to the next annulus.

   The following diagram shows the two bin numbering methods:



ccLabeledProjection::eRadial



ccLabeledProjection::eAngular

Parameters

radius	The radius of the circular projection model, in pixels.
centerOffset	The initial value for the image offset for the projection model.

rMin The inner radius for the projection model. All pixels inside the inner radius are set to the don't care value.

rMax The outer radius for the projection model. All pixels outside this radius are set to the don't care value. If you specify 0 for *rMax*, then the bins extend to edges of the projection model.

numAnnuli The number of annuli in this model. If you are creating a circular projection model, you should set this to be the same as *radius.*

numSectors The number of sectors in this model.

binOrder The bin ordering method. *binOrder* must be one of the following values:

ccLabeledProjection::eRadial
ccLabeledProjection::eAngular

orientation The bin ordering orientation. *orientation* must be one of the following values:

ccLabeledProjection::eClockwise
ccLabeledProjection::eCounterClockwise

startAngle The starting angle for the first sector, specified relative to the positive side of the client coordinates system x-axis.

endAngle The end angle for the last sector, specified relative to the positive side of the client coordinates system x-axis. Pixels between *endAngle* and *startAngle* are set to the don't care value.

Throws

ccLabeledProjection::BadParams
The parameters you specified produced a projection model with every pixel set to 0.

radius

```
c_Int32 radius() const;
```

Returns the radius of the projection model.

Throws

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

centerOffset

```
cc2Vect centerOffset()const;
```

Returns the center offset of the projection model.

Throws
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

centerPosition

```
cc2Vect centerPosition() const;
```

Returns the center position of the projection model.

Throws
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

rMin

```
double rMin() const;
```

Returns the minimum radius of the projection model.

Throws
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

rMax

```
double rMax() const;
```

Returns the maximum radius of the projection model.

Throws
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

numAnnuli

```
c_Int32 numAnnuli() const;
```

Returns the number of annuli in the projection model.

Throws
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

numSectors

```
c_Int32 numSectors() const;
```

Returns the number of sectors in the projection model.

Throws
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

orientation

```
enum ccLabeledProjection::Orientation orientation() const;
```

Returns the orientation of this circular projection model. The orientation determines whether angles increase in the clockwise or counter-clockwise direction and the order in which sectors are numbered. The value returned by this function is one of the following:

ccLabeledProjection::eClockwise
ccLabeledProjection::eCounterClockwise

Throws

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

startAngle

```
ccDegree startAngle() const;
```

Returns the start angle of the projection model.

Throws

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

endAngle

```
ccDegree endAngle() const;
```

Returns the end angle of the projection model.

Throws

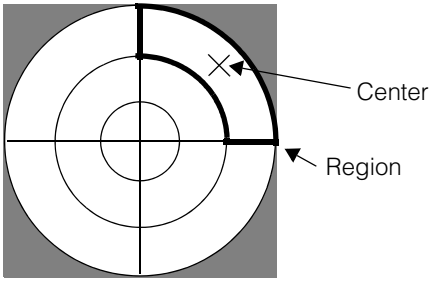
ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

binCenter

```
cc2Vect binCenter(int index) const;
cc2Vect binCenter(int annulus, int sector) const;
```

- `cc2Vect binCenter(int index) const;`

Returns the center of the region within this **ccCircularLabeledProjectionModel** that corresponds to the specified pixel in the projection image. The center of the region is defined as the point equidistant between the inner and outer boundary at the midpoint between the lower and higher edge of the region, as shown in the following figure:



The position is reported with sub-pixel accuracy in image coordinates.

Parameters

index The bin within the projection image.

Throws

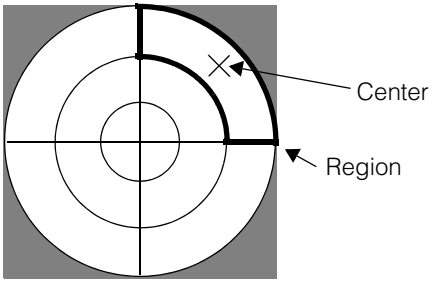
ccLabeledProjection::BadParams
index does not correspond to a bin.

ccLabeledProjection::BadSectorCount
This **ccCircularLabeledProjectionModel** has no sectors.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

- `cc2Vect binCenter(int annulus, int sector) const;`

  Returns the center of the specified region within this **ccCircularLabeledProjectionModel**. The center of the region is defined as the point equidistant between the inner and outer boundary at the midpoint between the lower and higher edge of the region, as shown in the following figure:



  The position is reported with sub-pixel accuracy in image coordinates.

  Parameters

  annulus The annulus number of the region

  sector The sector number of the region

  Throws

  ccLabeledProjection::BadParams
  annulus and *sector* do not specify a region in this **ccCircularLabeledProjectionModel**.

  ccLabeledProjection::NotTrained
  This **ccCircularLabeledProjectionModel** has not been trained.

sectorAngle `ccDegree sectorAngle() const;`

Returns the angle of the given sector, defined as the angle between the image coordinate system x-axis and a line drawn from the center of the model through the center of the sector.

Throws

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

ccLabeledProjection::BadParams
sector does not specify a sector.

sectorOrientation

```
ccDegree sectorOrientation(int sector) const;
```

Returns the orientation from the center of the **ccCircularLabeledProjectionModel** to the middle of the given sector.

Parameters

sector The sector

Throws

ccLabeledProjection::BadParams
sector does not correspond to a sector.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

angleSector

```
int angleSector(ccDegree angle) const;
```

Returns the number of the sector in this **ccCircularLabeledProjectionModel** through which a line drawn at the specified angle from the image coordinate system x-axis passes.

Parameters

angle The angle

Throws

ccLabeledProjection::BadParams
The line specified by *angle* does not pass through a sector.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

annulusWidth

```
double annulusWidth() const;
```

Returns the width of each annulus in image coordinate system units.

Throws

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

annulusDistance

```
double annulusDistance(int annulus) const;
```

Returns the distance from the center of this **ccCircularLabeledProjectionModel** to the center of the specified annulus.

Parameters

annulus The annulus

Throws

ccLabeledProjection::BadParams
annulus not specify an annulus.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

distanceAnnulus

```
int distanceAnnulus(double distance) const;
```

Returns the annulus which lies at the specified distance from the center of this **ccCircularLabeledProjectionModel**.

Parameters

distance The distance

Throws

ccLabeledProjection::BadParams
No annulus is present at *distance* from the center.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

sectorWindow

```
ccPelRect sectorWindow(int sector) const;
```

Returns a **ccPelRect** that describes the portion of the projection image that was formed by summing the annuli within the specified sector. The returned **ccPelRect** is in the image coordinate system of the projection image.

Parameters

sector The sector

Throws

ccLabeledProjection::WrongOrder
This **ccCircularLabeledProjectionModel**'s bin ordering method is angular.

ccLabeledProjection::BadParams
sector does not specify a sector.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

annulusWindow

```
ccPelRect annulusWindow(int annulus) const;
```

Returns a **ccPelRect** that describes the portion of the projection image that was formed by summing the sectors within the specified annulus. The returned **ccPelRect** is in the image coordinate system of the projection image.

Parameters

annulus The annulus

Throws

ccLabeledProjection::WrongOrder
This **ccCircularLabeledProjectionModel**'s bin ordering method is radial.

ccLabeledProjection::BadParams
annulus does not specify an annulus.

ccLabeledProjection::NotTrained
This **ccCircularLabeledProjectionModel** has not been trained.

ccClassifierFeatureScore

```
#include <ch_cvl/clsscore.h>

class ccClassifierFeatureScore: public ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

An abstract base class from which all Classifier scoring functions are derived. You should not instantiate objects of this type, nor should you attempt to derive your own scoring functions from this class.

Constructors/Destructors

ccClassifierFeatureScore

```
virtual ~ccClassifierFeatureScore();
```

Public Member Functions

score

```
virtual double score(double featureValue) const = 0;
```

Returns a score for the given feature value. The exact calculation depends upon the derived implementation. The returned score is normalized to a value between 0.0 and 1.0.

This function returns -1 to indicate that the score should be ignored (not included in the computation of the overall score for a feature vector).

Parameters

featureValue The value to score.

clone

```
virtual ccClassifierFeatureScore *clone() const=0;
```

Constructs a **ccClassifierFeatureScore** identical to this one.

ccClassifierFeatureScore

ccClassifierFeatureScoreIgnore

```
#include <ch_cvl/clsscore.h>

class ccClassifierFeatureScoreIgnore:
  public ccClassifierFeatureScore;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A scoring function that always returns -1.

Public Member Functions

score

```
virtual double score(double featureValue) const;
```

Returns a score of -1 which means that this score is ignored when the geometric mean is computed.

Parameters

featureValue The value not to score.

clone

```
virtual ccClassifierFeatureScore *clone() const;
```

Constructs a **ccClassifierFeatureScoreIgnore** identical to this one.

ccClassifierFeatureScoreIgnore

ccClassifierFeatureScoreOneSided

```
#include <ch_cvl/clsscore.h>

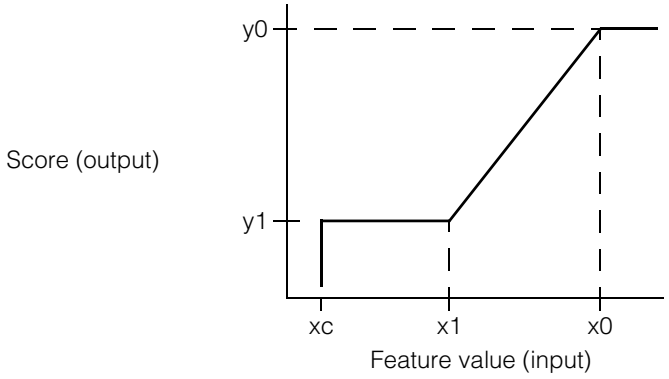
class ccClassifierFeatureScoreOneSided:
  public ccClassifierFeatureScore;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A scoring function that lets you configure a one-sided scoring function.

A one-sided scoring function maps a feature value to an output score using a function defined by the points shown in the following figure:



Constructors/Destructors

ccClassifierFeatureScoreOneSided

```
ccClassifierFeatureScoreOneSided();
```

Constructs a **ccClassifierFeatureScoreOneSided** with the default values shown in the **init** function signature.

Public Member Functions

init

```
void init (double x0 = 0.0, double x1 = 1.0,
  double xc = 1.0, double y0 = 1.0, double y1 = 0.0);
```

Initializes the breakpoints in this **ccClassifierFeatureScoreOneSided** to the supplied values.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be between 0.0 and 1.0.

---

x0

```
double x0 () const;
void x0 (double x0);
```

- `double x0 () const;`

  Returns the *x0* value.

- `void x0 (double x0);`

  Sets the *x0* value.

  Parameters

x0	The value to set.

---

x1

```
double x1 () const;
void x1 (double x1);
```

- `double x1 () const;`

  Returns the *x1* value.

- `void x1 (double x1);`

  Sets the *x1* value.

Parameters

x1 The value to set. *x1* must be between *xc* and *x0*.

xc

```
double xc () const;
void xc (double xc);
```

- `double xc () const;`

  Returns the *xc* value.

- `void xc (double xc);`

  Sets the *xc* value.

  Parameters

  xc The value to set.

y0

```
double y0 () const;
void y0 (double y0);
```

- `double y0 () const;`

  Returns the *y0* value.

- `void y0 (double y0);`

  Sets the *y0* value.

  Parameters

  y0 The value to set. *y0* must be between 0.0 and 1.0.

  Throws

  ccClassifierDefs::BadParams

  y0 is outside the range 0.0 to 1.0.

y1

```
double y1 () const;
void y1 (double y1);
```

- `double y1 () const;`

  Returns the *y1* value.

- `void y1 (double y1);`

  Sets the *y1* value.

  Parameters

  y1 The value to set. *y1* must be between 0.0 and 1.0.

  Throws

  ccClassifierDefs::BadParams
  y1 is outside the range 0.0 to 1.0.

score `virtual double score(double featureValue) const;`

Returns the score for the given feature value.

Parameters

featureValue The value to score.

clone `virtual ccClassifierFeatureScore *clone() const;`

Constructs a **ccClassifierFeatureScoreOneSided** identical to this one.

ccClassifierFeatureScoreTwoSided

```
#include <ch_cvl/clsscore.h>

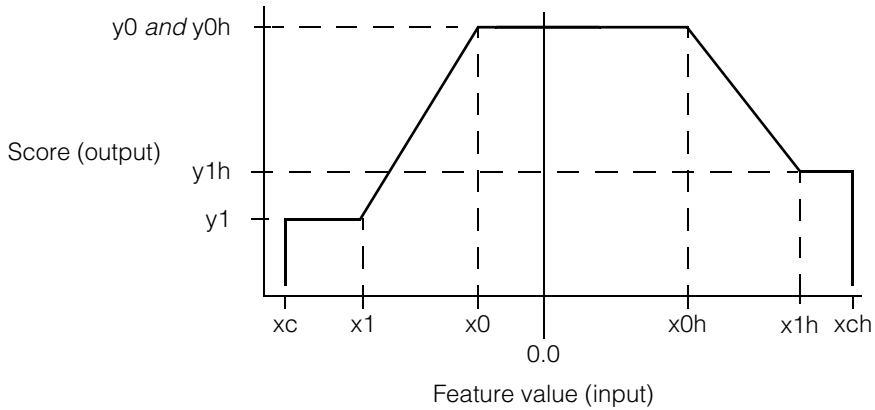
class ccClassifierFeatureScoreTwoSided:
  public ccClassifierFeatureScore;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A scoring function which lets you configure a two-sided scoring function.

A two-sided scoring function maps a feature value to an output score using a function defined by the points shown in the following figure:



A two-sided scoring function is implemented as two one-sided scoring functions.

Constructors/Destructors

ccClassifierFeatureScoreTwoSided

```
ccClassifierFeatureScoreTwoSided();
```

Constructs a **ccClassifierFeatureScoreTwoSided** with the default values shown in the **init** function signature.

Public Member Functions

init

```
void init (double x0 = 0.0, double x1 = -1.0,
  double xc = -1.0, double x0h = 0.0,
  double x1h = 1.0, double xch = 1.0, double y0 = 1.0,
  double y1 = 0.0, double y0h = 1.0, double y1h = 0.0);
```

Initializes the breakpoints in this **ccClassifierFeatureScoreTwoSided** to the supplied values.

Parameters

x0	The value for the *x0* point
x1	The value for the *x1* point
xc	The value for the *xc* point
x0h	The value for the *x0h* point
x1h	The value for the *x1h* point
xch	The value for the *xch* point
y0	The value for the *y0* point
y1	The value for the *y1* point
y0h	The value for the *y0h* point
y1h	The value for the *y1h* point

---

x0

```
double x0 () const;
void x0 (double x0);
```

- `double x0 () const;`

  Returns the *x0* value.

- `void x0 (double x0);`

  Sets the *x0* value.

  Parameters

x0	The value to set. *x0* should be less than *x0h*, *x1h*, and *xch*.

x1

```
double x1 () const;
void x1 (double x1);
```

- `double x1 () const;`

  Returns the *x1* value.

- `void x1 (double x1);`

  Sets the *x1* value.

  Parameters

x1	The value to set. *x1* should be between *xc* and *x0* and less than *x0h*, *x1h*, and *xch*.

xc

```
double xc () const;
void xc (double xc);
```

- `double xc () const;`

  Returns the *xc* value.

- `void xc (double xc);`

  Sets the *xc* value.

  Parameters

xc	The value to set. *xc* should be less than *x0h*, *x1h*, and *xch*.

x0h

```
double x0h () const;
void x0h (double x0h);
```

- `double x0h () const;`

  Returns the *x0h* value.

- `void x0h (double x0h);`

  Sets the *x0h* value.

Parameters

x0h	The value to set. *x0h* should be greater than *xc*, *x1*, and *x0*.

x1h

```
double x1h () const;
void x1h (double x1h);
```

- `double x1h () const;`

  Returns the *x1h* value.

- `void x1h (double x1h);`

  Sets the *x1h* value.

  Parameters

x1h	The value to set. *x1h* should be between *x0h* and *xch* and be greater than *xc*, *x1*, and *x0*.

xch

```
double xch () const;
void xch (double xch);
```

- `double xch () const;`

  Returns the *xch* value.

- `void xch (double xch);`

  Sets the *xch* value.

  Parameters

xch	The value to set. *xch* should be greater than *xc*, *x1*, and *x0*.

y0

```
double y0 () const;
void y0 (double y0);
```

- `double y0 () const;`

  Returns the *y0* value.

- `void y0 (double y0);`

  Sets the *y0* value.

  Parameters

  y0 The value to set. *y0* must be between 0.0 and 1.0.

  Throws

  ccClassifierDefs::BadParams
  y0 is outside the range 0.0 to 1.0.

y1

```
double y1 () const;
void y1 (double y1);
```

- `double y1 () const;`

  Returns the *y1* value.

- `void y1 (double y1);`

  Sets the *y1* value.

  Parameters

  y1 The value to set. *y1* must be between 0.0 and 1.0.

  Throws

  ccClassifierDefs::BadParams
  y1 is outside the range 0.0 to 1.0.

y0h

```
double y0h () const;
void y0h (double y0h);
```

- `double y0h () const;`

  Returns the *y0h* value.

- `void y0h (double y0h);`

  Sets the *y0h* value.

  Parameters

  y0h The value to set. *y0h* must be between 0.0 and 1.0.

  Throws

ccClassifierDefs::BadParams
y0h is outside the range 0.0 to 1.0.

y1h

```
double y1h () const;
void y1h (double y1h);
```

- `double y1h () const;`

  Returns the *y1h* value.

- `void y1h (double y1h);`

  Parameters
  y1h The value to set. *y1h* must be between 0.0 and 1.0.

  Throws
  ccClassifierDefs::BadParams
  y1h is outside the range 0.0 to 1.0.

Low

```
const ccClassifierFeatureScoreOneSided &Low() const;
```

Returns the **ccClassifierFeatureScoreOneSided** object that makes up the low side of this **ccClassifierFeatureScoreTwoSided**.

High

```
const ccClassifierFeatureScoreOneSided &High() const;
```

Returns the **ccClassifierFeatureScoreOneSided** object that makes up the high side of this **ccClassifierFeatureScoreTwoSided**.

score

```
virtual double score(double featureValue) const;
```

Returns the score for the given feature value.

Parameters
featureValue The value to score.

clone

```
virtual ccClassifierFeatureScore *clone() const;
```

Constructs a **ccClassifierFeatureScoreTwoSided** identical to this one.

ccClassifierFeatureVector

```
#include <ch_cvl/clssfier.h>

class ccClassifierFeatureVector;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that defines a single feature vector. A feature vector represents a collection of measurements of a single object or feature.

Constructors/Destructors

ccClassifierFeatureVector

```
ccClassifierFeatureVector(
  cmStd vector<double> &featureValues);
```

Constructs a **ccClassifierFeatureVector** using the supplied values.

Parameters

featureValues A vector of values.

Public Member Functions

featureValues

```
const cmStd vector<double> &featureValues();
```

Returns the feature values used to construct this **ccClassifierFeatureVector**.

ccClassifierFeatureVector

ccClassifierRule

```
#include <ch_cvl/clssfier.h>

class ccClassifierRule;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds a single Classifier tool rule. A rule contains a vector of scoring methods, one method for each dimension of the feature vectors being scored.

Constructors/Destructors

ccClassifierRule

```
ccClassifierRule();
```

Constructs a **ccClassifierRule** with a label value of 0 and no scoring methods.

Public Member Functions

init

```
void init(
  cmStd vector<ccClassifierFeatureScorePtrh>
  &scoringMethods, c_Int32 label);
```

Initializes this **ccClassifierRule** using the supplied vector of scoring methods and the supplied label.

Parameters

scoringMethods A vector of scoring methods. The elements of *scoringMethods* must be **ccPointerHandle** objects derived from the **ccClassifierFeatureScore** class. The **ccClassifierFeatureScorePtrh** type is a typedef defined in *clsscore.h.*

label The label for this rule. This value is returned by the Classifier tool when it classifies a feature vector.

Throws

ccClassifierDefs::BadParams

label is less than 0 or *scoringMethods* has no elements.

score

```
double score(
  const ccClassifierFeatureVector &featureVector) const;
```

Computes the overall score for the supplied **ccClassifierFeatureVector** using the scoring methods that comprise this rule.

The overall score is the geometric mean of the scores returned by the individual scoring methods (ignoring any scores of -1).

Parameters

featureVector The feature vector to score. *featureVector* must have the same number of elements as there are scoring methods in this **ccClassifierRule**.

Throws

ccClassifierDefs::BadParams

featureVector does not contain the same number of elements as this **ccClassifierRule** contains scoring methods; one or more of the one-sided scoring functions contained in this **ccClassifierRule** do not meet the requirement that *x1* must be between *x0* and *xc*; or one or more of the two-sided scoring functions contained in this **ccClassifierRule** do not meet the requirement that *x1* must be between *x0* and *xC* and *x1h* must be between *x0h* and *xch*.

scoringMethods

```
const cmStd vector<ccClassifierFeatureScorePtrh>
  &scoringMethods() const;
```

Returns the vector of scoring methods defined for this **ccClassifierRule**.

label

```
c_Int32 label() const;
```

Return the label of this **ccClassifierRule**.

ccClassifierRuleTable

```
#include <ch_cvl/clssfier.h>

class ccClassifierRuleTable;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds a rule table. A rule table is a vector of rules, one for each category into which you are classifying feature vectors.

Public Member Functions

train

```
void train(const cmStd vector<ccClassifierRule> &rules);
```

Initializes this **ccClassifierRuleTable** using the supplied vector of rules.

Parameters

rules A vector of **ccClassifierRule**s. You should supply as many elements as you have classifications. Each of the **ccClassifierRule**s in *rules* must be the same size.

Throws

ccClassifierDefs::BadParams
rules has no elements or the elements of *rules* are not all the same size.

classifyHardThreshold

```
c_Int32 classifyHardThreshold (
  const ccClassifierFeatureVector &featureVector,
  double hardThreshold) const;
```

Classifies the supplied feature vector. This function computes the score for the supplied feature vector using each rule contained in this **ccClassifierRuleTable**. The function returns the label of the first rule that produces a score greater than or equal to the supplied threshold.

If no rule produces a score greater than the threshold, the function returns -1.

Parameters

featureVector The feature vector to classify. *featureVector* must have the same number of elements that each of the rules in this **ccClassifierRuleTable** has.

hardThreshold The score threshold.

Throws

ccClassifierDefs::BadParams
hardThreshold is less than 0.0 or greater than 1.0; *featureVector* does not have the same number of elements as the first rule in this **ccClassifierRuleTable**; one or more of the one-sided scoring functions contained in one of this **ccClassifierRuleTable**'s rules do not meet the requirement that *x1* must be between *x0* and *xc*; or one or more of the two-sided scoring functions contained in one of this **ccClassifierRuleTable**'s rules do not meet the requirement that *x1* must be between *x0* and *xC* and *x1h* must be between *x0h* and *xch*.

ccClassifierDefs::NotTrained
This **ccClassifierRuleTable** has not been trained (no rules have been supplied).

rules

```
const cmStd vector<ccClassifierRule> &rules() const;
```

Returns the rules contained within this **ccClassifierRuleTable**.

ccCnlSearchDefs

```
#include <cnlsrch.h>

class ccCnlSearchDefs;
```

A name space that holds enumerations and constants used with CNLSearch.

Enumerations

Accuracy

`enum Accuracy`

This enumeration defines CNLSearch accuracy levels.

Value	Meaning
eVeryFine	Specifies highest-accuracy search
eFine	Specifies intermediate-accuracy search
eCoarse	Specifies coarse search
kDefaultAccuracy	The default accuracy, as defined in *cnlsearch.h*

Notes

You can only search at accuracies which were trained. Use the **ccCnlSearchTrainParams::accuracies()** function to specify the accuracies for training.

Algorithm

`enum Algorithm`

This enumeration defines the CNLSearch algorithms.

Value	Meaning
eNormalizedCnlpas	Specifies linear mode search using a conservative search strategy and normalized scoring. Correlation coefficients less than 0 produce scores of 0.0.
eAbsoluteCnlpas	Same as *eNormalizedCnlpas* except that absolute scoring is used. Both negative and positive correlation coefficients produce positive scores.

Value	Meaning
eNonlinearCnlpas	Specifies nonlinear mode search.
eNormalizedSearch	Specifies linear-mode search using an aggressive search strategy and normalized scoring. Correlation coefficients less than 0 produce scores of 0.0.
eAbsoluteSearch	Same as *eNormalizedSearch* except that absolute scoring is used. Both negative and positive correlation coefficients produce positive scores.
kDefaultAlgorithm	Default search algorithm, as defined in *cnlsrch.h*.

Notes

These enumerations may be OR'd together when you train a model. When you search for a model, you can specify only an algorithm for which the model has been trained.

You can train absolute and normalized versions of an algorithm with no memory or performance cost at training time or run time. Aside from this, training a model for additional algorithms takes more time and requires more memory for the model.

DrawMode

```
enum DrawMode;
```

This enumeration defines the types of result graphics you can draw for CNLSearch results.

Value	Meaning
eDrawOrigin	Draws a cross at the location of the model origin.
eDrawLabel	Labels the model origin with a text label.
eDrawBoundingBox	Draws a box around the model area.
eDrawStandard	Draws a labeled cross at the model origin and a box around the model area.

Constants

The following constants provide default values for certain training-time and run-time parameters:

Value	Meaning
kDefaultAcceptTimes1000	An integer containing the default acceptance threshold times 1000.
kDefaultConfusionTimes1000	An integer containing the default confusion threshold times 1000.
kDefaultMaxNumResults	An integer containing the default value for the number of results.
kDefaultLowThreshold	An integer containing the default edge detection low threshold value.
kDefaultHighThreshold	An integer containing the default edge detection high threshold value.
kMaxModelWidth	An integer specifying the maximum width, in pixels, of a model training image.
kMaxModelPelCount	An integer specifying the maximum size, in pixels, of a model training image.

ccCnlSearchDefs

ccCnlSearchModel

```
#include <ch_cvl/cnlsrch.h>

class ccCnlSearchModel: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that contains a CNLSearch model. A **ccCnlSearchModel** contains all of the information required to search for a pattern in an image.

Constructors/Destructors

ccCnlSearchModel

```
ccCnlSearchModel();

virtual ~ccCnlSearchModel();

ccCnlSearchModel(const ccCnlSearchModel& src);
```

- `ccCnlSearchModel();`

  Constructs a model. The model contains no training information but does have an origin of (0.0, 0.0).

Public Member Functions

name

```
ccCvlString name() const;

void name(const ccCvlString& name);
```

- `ccCvlString name() const;`

  Return the name of this model.

- `void name(const ccCvlString& name);`

  Set the name of this model. The model name is provided as a user convenience only.

  Parameters

  name — A const string reference containing the name to assign to this **ccCnlSearchModel**.

origin

```
cc2Vect origin() const;
void origin(const cc2Vect& origin);
```

- `cc2Vect origin() const;`

  Return the model origin.

- `void origin(const cc2Vect& origin);`

  Set the model origin. The origin must be specified in the client coordinate system of the image supplied to the **train()** member function. The origin is a point attached to the model, but it may reside outside of the model. When an instance of the model is found, the result location is defined by the position of the origin in the search image.

  Parameters

  origin — A reference to a const **cc2Vect** containing the origin of this model.

  Notes

  You can set and get a model's origin whether the model is trained or untrained. Training a model does not affect its origin.

train

```
void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccCnlSearchTrainParams& params,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccCnlSearchTrainParams& params,
  const ccPelRect& partialMatchRange, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccCnlSearchTrainParams& params,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccCnlSearchTrainParams& params,
  const ccPelRect& partialMatchRange, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0);
```

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const ccCnlSearchTrainParams& params,
    ccDiagObject* dObj=0, c_UInt32 dFlags=0);
  ```

  Train this **ccCnlSearchModel** using the supplied model image and model training parameters. If the model is already trained, the existing training information is discarded.

  Parameters

  image — A **const** reference to an image that contains the pattern to train as a model. *image* must be at least 8 x 8 pixels in size.

  params — A **ccCnlSearchTrainParams** that contains the parameters to use to train this model.

  dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff
  ```

Throws

ccCnlSearchDefs::BadParams
params.algorithms specifies a nonlinear algorithm but *params.lowThreshold* is greater than *params.highThreshold*.

ccCnlSearchDefs::BadImageSize
image is too small.

ccCnlSearchDefs::BadImageContent
image does not contain enough information to create a reliable model.

ccCnlSearchDefs::BadImage
image is unbound.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const ccCnlSearchTrainParams& params,
    const ccPelRect& partialMatchRange, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0);
  ```

Train this **ccCnlSearchModel** using the supplied model image, model training parameters, and partial match search range. If the model is already trained, the existing training information is discarded.

The partial match search range is only used if you have selected advanced training mode. In general, you should specify the same region that you expect to specify at run time. See the function **maxPartialMatchRange()** for a description of how the partial match search range is specified.

Parameters

image A **const** reference to an image that contains the pattern to train as a model. *image* must be at least 8 x 8 pixels in size.

params A **ccCnlSearchTrainParams** that contains the parameters to use to train this model.

partialMatchRange
An **ccPelRect** that specifies the amount by which a model instance may lie outside the search image and still be found.

The value you supply for *partialMatchRange* is clipped to the value you supplied to **maxPartialMatchRange()**.

dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:
  ```

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccCnlSearchDefs::BadParams
params.algorithms specifies a nonlinear algorithm but *params.lowThreshold* is greater than *params.highThreshold*.

ccCnlSearchDefs::BadImageSize
image is too small.

ccCnlSearchDefs::BadImageContent
image does not contain enough information to create a reliable model.

ccCnlSearchDefs::BadImage
image is unbound.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccCnlSearchTrainParams& params,
    ccDiagObject* dObj=0, c_UInt32 dFlags=0);
  ```

Train this **ccCnlSearchModel** using the supplied model image, model training parameters, and mask image. If the model is already trained, the existing training information is discarded.

Parameters

image — A **const** reference to an image that contains the pattern to train as a model. *image* must be at least 8 x 8 pixels.

mask — A **const** reference to an image containing a mask to use when training this model. *mask* must have the same dimensions as *image*. Only pixels in *image* which correspond to pixels in *mask* that have zero values are included in the trained model. Nonzero pixels are treated as don't care pixels.

params — A **ccCnlSearchTrainParams** that contains the parameters to use to train this model.

dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
  ```

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccCnlSearchDefs::BadParams
params.algorithms specifies a nonlinear algorithm but *params.lowThreshold* is greater than *params.highThreshold*.

ccCnlSearchDefs::BadImageSize
image is too small.

ccCnlSearchDefs::BadMaskSize
mask is not the same size as *image*.

ccCnlSearchDefs::BadImageContent
image does not contain enough information to create a reliable model.

ccCnlSearchDefs::BadImage
image is unbound.

ccCnlSearchDefs::NotImplemented
A mask was supplied but the algorithms included one of the following:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas

Notes

If *mask* is unbound, the model is trained as if no mask had been supplied; no exception is thrown.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccCnlSearchTrainParams& params,
    const ccPelRect& partialMatchRange, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0);
  ```

  Train this **ccCnlSearchModel** using the supplied model image, model training parameters, mask image, and partial match search range. If the model is already trained, the existing training information is discarded.

  The partial match search range is only used if you have selected advanced training mode. In general, you should specify the same region that you expect to specify at run time. See the function **maxPartialMatchRange()** for a description of how the partial match search range is specified.

  Parameters

  image — A **const** reference to an image that contains the pattern to train as a model. *image* must be at least 8 x 8 pixels in size.

  mask — A **const** reference to an image containing a mask to use when training this model. *mask* must have the same dimensions as *image*. Only pixels in *image* which correspond to pixels in *mask* that have zero values are included in the trained model. Nonzero pixels are treated as don't care pixels.

  params — A **ccCnlSearchTrainParams** that contains the parameters to use to train this model.

  partialMatchRange
  An **ccPelRect** that specifies the amount by which a model instance may lie outside the search image and still be found.

  The value you supply for *partialMatchRange* is clipped to the value you supplied to **maxPartialMatchRange()**.

  dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff
  ```

Throws

ccCnlSearchDefs::BadParams
params.algorithms specifies a nonlinear algorithm but *params.lowThreshold* is greater than *params.highThreshold*.

ccCnlSearchDefs::BadImageSize
image is too small.

ccCnlSearchDefs::BadMaskSize
mask is not the same size as *image*.

ccCnlSearchDefs::BadImageContent
image does not contain enough information to create a reliable model.

ccCnlSearchDefs::BadImage
image is unbound.

ccCnlSearchDefs::NotImplemented
A mask was supplied but the algorithms included one of the following:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas

Notes

If *mask* is unbound, the model is trained as if no mask had been supplied; no exception is thrown.

transmitPveModel

```
void transmitPveModel(const ccCvlString& fname,
 c_UInt32 depth = 8);

void transmitPveModel(cmStd ostream& out,
 c_UInt32 depth = 8);
```

- ```
  void transmitPveModel(const ccCvlString& fname,
   c_UInt32 depth = 8);
  ```

Writes the CnlSearch model data to the specified file in PVE format, using *depth* bits. The bit depth must be less than or equal to 8 bits.

Parameters

fname	The name of the file to write the model data to.
depth	The bits per pixel of the stored model data.
  ```

Throws

ccCnlSearchDefs::NotTrained
The model is not trained.

ccCnlSearchDefs::NotImplemented
The model was trained with one of the following algorithms:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas

depth was greater than 8 bits.

ccCnlSearchDefs::TransmitErr
An error occurred writing the PVE data to the file.

ccCnlSearchDefs::BadPveModelSize
The model image does not conform to PVE model size restrictions.

ccCnlSearchDefs::BadImage
The training image was more than *depth* bits deep.

- ```
  void transmitPveModel(cmStd ostream& out,
    c_UInt32 depth = 8);
  ```

Writes the CnlSearch model data to the specified stream in PVE format, using *depth* bits. The bit depth must be less than or equal to 8 bits.

Parameters

out The stream to write the model data to.

depth The bits per pixel of the stored model data.

Throws

ccCnlSearchDefs::NotTrained
The model is not trained.

ccCnlSearchDefs::NotImplemented
The model was trained with one of the following algorithms:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas

depth was greater than 8 bits.

ccCnlSearchDefs::TransmitErr
An error occurred writing the PVE data to the stream.
  ```

ccCnlSearchDefs::BadPveModelSize
The model image does not conform to PVE model size restrictions.

ccCnlSearchDefs::BadImage
The training image was more than *depth* bits deep.

unTrain `void unTrain();`

Untrain this model. All training information is discarded and all memory associated with the training information is freed. The model origin is not affected.

Throws
ccCnlSearchDefs::NotTrained
This model is not currently trained.

isTrained `bool isTrained() const;`

Return *true* if this model is currently trained, *false* otherwise.

trainParams `ccCnlSearchTrainParams trainParams() const;`

Return the **ccCnlSearchTrainParams** used to train this model.

Throws
ccCnlSearchDefs::NotTrained
This model is not currently trained.

modelWidth `c_Int16 modelWidth() const;`

Return the width (in pixels) of the model image used to train this model.

Throws
ccCnlSearchDefs::NotTrained
This model is not currently trained.

modelHeight `c_Int16 modelHeight() const;`

Return the height (in pixels) of the model image.

Throws
ccCnlSearchDefs::NotTrained
This model is not currently trained.

trainImage

```
ccPelBuffer_const<c_UInt8> trainImage() const;
```

Return the image used to train this model.

Throws

ccCnlSearchDefs::NotTrained
This model is not currently trained.

trainEdgeImage

```
ccPelBuffer_const<c_UInt8> trainEdgeImage() const;
```

Return an image containing the edge map generated from the model image.

Throws

ccCnlSearchDefs::NotTrained
ccCnlsSearchDefs::eNonlinearCnlpas was not among the algorithms used to train this model.

trainMask

```
ccPelBuffer_const<c_UInt8> trainMask() const;
```

Return an image containing the mask used to train this model.

Throws

ccCnlSearchDefs::NotTrained
This model is not currently trained, or was not trained with a mask image.

minSearchImageSize

```
ccIPair minSearchImageSize(ccCnlSearchDefs::Algorithm alg)
  const;
```

Returns the size in pixels of smallest image which can be searched with this model using the specified algorithm. The minimum size is typically slightly larger than the dimensions of the training image.

Parameters

alg The algorithm for which to return the minimum model size. *alg* must be one of the following values:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas
ccCnlSearchDefs::eNormalizedSearch
ccCnlSearchDefs::eAbsoluteSearch
ccCnlSearchDefs::kDefaultAlgorithm

For more information, see the description of *ccCnlSearchDefs* on page 973.

Throws

ccCnlSearchDefs::NotTrained
This model is not currently trained.

---

run

```
void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccCnlSearchRunParams& params,
  ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0) const;

void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccCnlSearchRunParams& params,
  const ccPelRect& partialMatchRange,
  ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0) const;

void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccIPair& point, const ccCnlSearchRunParams& params,
  ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0) const;

void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccIPair& point, const ccCnlSearchRunParams& params,
  const ccPelRect& partialMatchRange,
  ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0) const;
```

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccCnlSearchRunParams& params,
    ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0) const;
  ```

  Search for this model within the supplied image. You supply a **ccCnlSearchRunParams** containing parameters for the search and a **ccCnlSearchResultSet** in which to store the results. The results are stored in decreasing order of score.

  Parameters

image	The image to search for the model.
params	The search parameters.
resultSet	A **ccCnlSearchResultSet** in which to store the results. *resultSet* is cleared before the search results are stored in it.
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
  ```

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccCnlSearchDefs::BadImageSize
image is too small to fully contain the model.

ccCnlSearchDefs::NotTrained
This model is not currently trained; the model was not trained at the accuracy level that you specified for the search; or the model was not trained for the algorithm specified for the search.

ccCnlSearchDefs::BadParams
params.confusion is less than *params.accept*; *params.maxNumResults* is less than or equal to 0; or *params.algorithm* is *ccCnlSearchDefs::eNonlinearCnlpas* and *params.lowThreshold* is greater than *params.highThreshold*.

ccCnlSearchDefs::BadImage
image is unbound.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccCnlSearchRunParams& params,
    const ccPelRect& partialMatchRange,
    ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0) const;
  ```

Search for this model within the supplied image. You supply a **ccCnlSearchRunParams** containing parameters for the search, a **ccPelRect** specifying a partial match search range, and a **ccCnlSearchResultSet** in which to store the results. The results are stored in decreasing order of score.

See the function **maxPartialMatchRange()** for a description of how the partial match search range is specified.

Parameters

image The image to search for the model.

params The search parameters.
  ```

partialMatchRange
: An **ccPelRect** that specifies the amount by which a model instance may lie outside the search image and still be found.

  The value you supply for *partialMatchRange* is clipped to the value you supplied to **maxPartialMatchRange()**.

  If you specify a range of **ccPelRect(0, 0, 1, 1)**, then no partial matches are allowed.

resultSet
: A **ccCnlSearchResultSet** in which to store the results. *resultSet* is cleared before the search results are stored in it.

dObj
: An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags
: The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

Throws

ccCnlSearchDefs::BadImageSize
: *image* is too small to fully contain the model.

ccCnlSearchDefs::NotTrained
: This model is not currently trained; the model was not trained at the accuracy level that you specified for the search; or the model was not trained for the algorithm specified for the search.

ccCnlSearchDefs::BadParams
: *params.confusion* is less than *params.accept*; *params.maxNumResults* is less than or equal to 0; *params.algorithm* is *ccCnlSearchDefs::eNonlinearCnlpas* and *params.lowThreshold* is greater than *params.highThreshold*; or *partialMatchRange* does not contain the point 0,0.

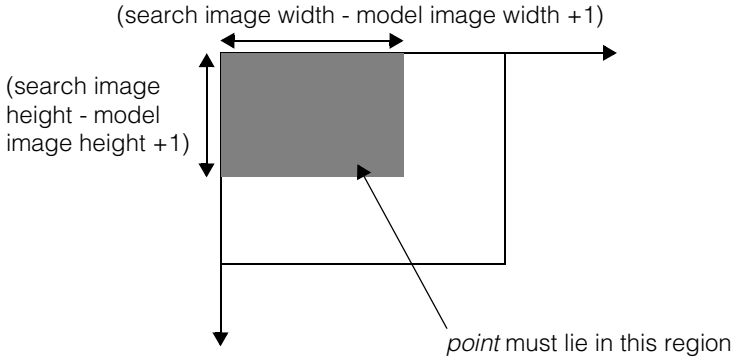
ccCnlSearchDefs::BadImage
: *image* is unbound.

ccCnlSearchDefs::NotImplemented
partialMatchRange is not (0, 0, 1, 1) and *params.algorithm* is not *ccCnlSearchDefs::eNormalizedSearch* or *ccCnlSearchDefs::eAbsoluteSearch*.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccIPair& point, const ccCnlSearchRunParams& params,
    ccCnlSearchResult& result, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0) const;
  ```

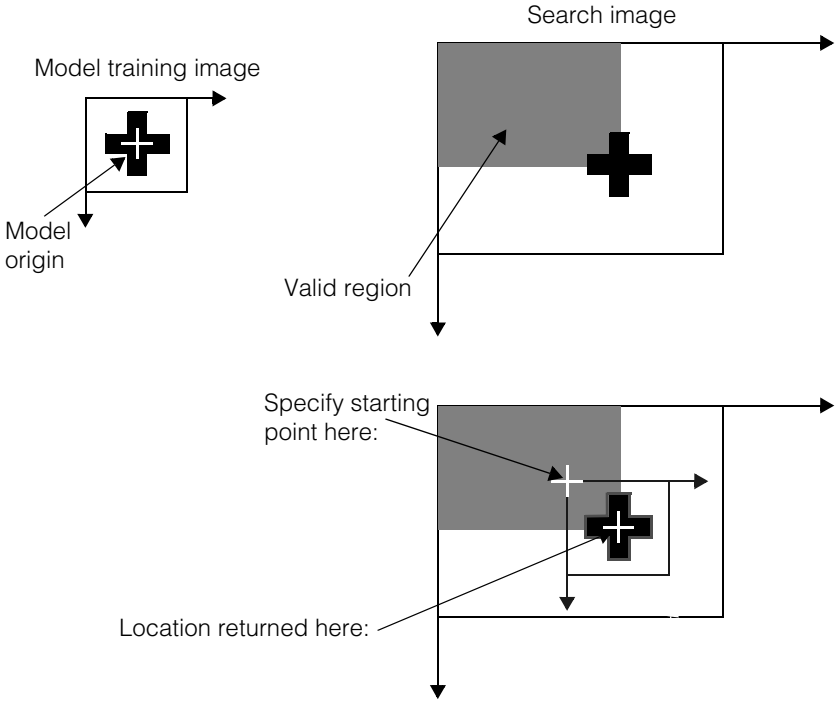
  Search for a single instance of this model within the supplied image starting at the supplied point. If the starting point you specify is close to the actual location of the model in the search image, this *point search* can be faster than a search that considers the entire area of the search image.

  The initial point, specified in whole pixel units relative to the origin of the search image, must lie within the following region of the search image:



  The search will be confined to the immediate vicinity of the point you specify. The point you specify corresponds to the upper-left corner of the model image. The model location
  ```

is returned, as usual, in terms of the model origin, as shown in the following figure:



You supply a **ccCnlSearchRunParams** containing parameters for the search and a **ccCnlSearchResult** in which to store the result.

Notes

Point search is only supported for the *ccCnlSearchDefs::eAbsoluteSearch* and *ccCnlSearchDefs::eNormalizedSearch* algorithms.

Parameters

image	The image to search for the model.
params	The search parameters.
result	A **ccCnlSearchResult** in which to store the result. *result* is cleared before the search result is stored in it.
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccCnlSearchDefs::BadImageSize
image is too small to fully contain the model.

ccCnlSearchDefs::BadImage
image is unbound.

ccCnlSearchDefs::NotTrained
This model is not currently trained; the model was not trained at the accuracy level that you specified for the search; or the model was not trained for the algorithm specified for the search.

ccCnlSearchDefs::BadParams
params.algorithm is neither *ccCnlSearchDefs::eAbsoluteSearch* nor *ccCnlSearchDefs::eNormalizedSearch*.

ccSearch::BadPoint
point does not lie within the valid region.

Notes

image must be bound.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccIPair& point, const ccCnlSearchRunParams& params,
    const ccPelRect& partialMatchRange,
    ccCnlSearchResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0) const;
  ```

  Search for a single instance of this model within the supplied image starting at the supplied point. If the starting point you specify is close to the actual location of the model in the search image, this *point search* can be faster than a search that considers the entire area of the search image.

  See the previous overload of **run()** for a description of how the starting point is specified.

  You supply a **ccCnlSearchRunParams** containing parameters for the search, a **ccPelRect** specifying a partial match search range, and a **ccCnlSearchResult** in which to store the result.

  See the function **maxPartialMatchRange()** for a description of how the partial match search range is specified.

  Notes

  Point search is only supported for the *ccCnlSearchDefs::eAbsoluteSearch* and *ccCnlSearchDefs::eNormalizedSearch* algorithms.

  Parameters

  image — The image to search for the model.

  params — The search parameters.

  partialMatchRange — An **ccPelRect** that specifies the amount by which a model instance may lie outside the search image and still be found.

  The value you supply for *partialMatchRange* is clipped to the value you supplied to **maxPartialMatchRange()**.

  If you specify a range of **ccPelRect(0, 0, 1, 1)**, then no partial matches are allowed.

  result — A **ccCnlSearchResult** in which to store the result. *result* is cleared before the search result is stored in it.

  dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:
  ```

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccCnlSearchDefs::BadImageSize
image is too small to fully contain the model.

ccCnlSearchDefs::BadImage
image is unbound.

ccCnlSearchDefs::NotTrained
This model is not currently trained; the model was not trained at the accuracy level that you specified for the search; or the model was not trained for the algorithm specified for the search.

ccCnlSearchDefs::BadParams
params.algorithm is neither *ccCnlSearchDefs::eAbsoluteSearch* nor *ccCnlSearchDefs::eNormalizedSearch* or *partialMatchRange* does not contain the point 0,0.

ccSearch::BadPoint
point does not lie within the valid region.

Notes

image must be bound.

resultStatistics

```
void resultStatistics(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccCnlSearchResult& result, double* sumImage,
  double* sumSquareImage, double* sumImageTimesModel,
  double* contrast);
```

Return information about the pixel values for a model image-sized location within a search image. The model location is rounded to the nearest whole pixel location and the following quantities are computed:

- The sum of all image pixels which correspond to care pixels in the model image
- The sum of the squares of all image pixels which correspond to care pixels in the model image
- The sum of the products of image pixels and model pixels, for all image pixels which correspond to care pixels in the model image

- The ratio of the standard deviation of the image pixels to the standard deviation of the model pixels, for all image pixels which correspond to care pixels in the model image

Parameters

image The image for which these statistics are computed

result A single **ccCnlSearchResult** describing the location within the *image* at which the statistics are to be computed

sumImage A pointer to a **double** into which the sum of all of the pixels in the image that correspond to care pixels in the model is copied

sumSquareImage
A pointer to a **double** into which the sum of the squares of all of the pixels in the image that correspond to care pixels in the model is copied

sumImageTimesModel
A pointer to a **double** into which the sum of the products of each pair of model and image pixels that correspond to care pixels in the model is copied

contrast A pointer to a **double** into which the ratio of the standard deviation of pixel values in the image to the standard deviation of pixel values in the model, for all pixels corresponding to care pixels in the model, is copied

Notes

If this model was trained using a mask, only pixels corresponding to care pixels in the model are included when the statistics are computed.

Throws

ccCnlSearchDefs::BadImageSize
The image is too small to contain the model.

ccCnlSearchDefs::NotTrained
This model is not trained.

---

saveImage

```
bool saveImage() const;
void saveImage(bool save);
```

- `bool saveImage() const;`

This function returns whether or not this **ccCnlSearchModel** is configured to save the image pixels used for training.

- `void saveImage(bool save);`

  Sets whether or not a copy of the image pixels used to train this **ccCnlSearchModel** are to be saved at training. If you call this function with an argument of false after training any saved training image pixels are discarded.

  Parameters

  save — If true, the pixels are saved at training. If false, the pixels are not saved.

  Throws

  ccCnlSearchDefs::BadParams — This function was called with *save* set to true after this **ccCnlSearchModel** was trained (and no pixels had been saved when it was trained).

  Notes

  You cannot archive a trained **ccCnlSearchModel** unless this function had been called with an argument of true prior to training. (To archive a trained **ccCnlSearchModel**, the training image pixels must be available.)

maxPartialMatchRange

```
ccPelRect maxPartialMatchRange() const;

void maxPartialMatchRange(const ccPelRect& range);
```

- `ccPelRect maxPartialMatchRange() const;`

  This function returns the maximum partial match search range that can be specified for searches using this **ccCnlSearchModel**.

- `void maxPartialMatchRange(const ccPelRect& range);`

  This function sets the maximum partial match search range that can be specified for searches using this **ccCnlSearchModel**. When you specify a partial match search range for a particular search, the range is always clipped to the maximum value you specify with this function.

  The partial match search range is specified as a **ccPelRect**. The **ccPelRect** defines the tolerance of this **ccCnlSearchModel** for finding model instances that are partially outside of the search image. The **ccPelRect** is interpreted as a dilation region that is applied to the search image to produce the region in which matches are allowed. The simplest way to specify the partial match search range is to construct a **ccPelRect** as follows:

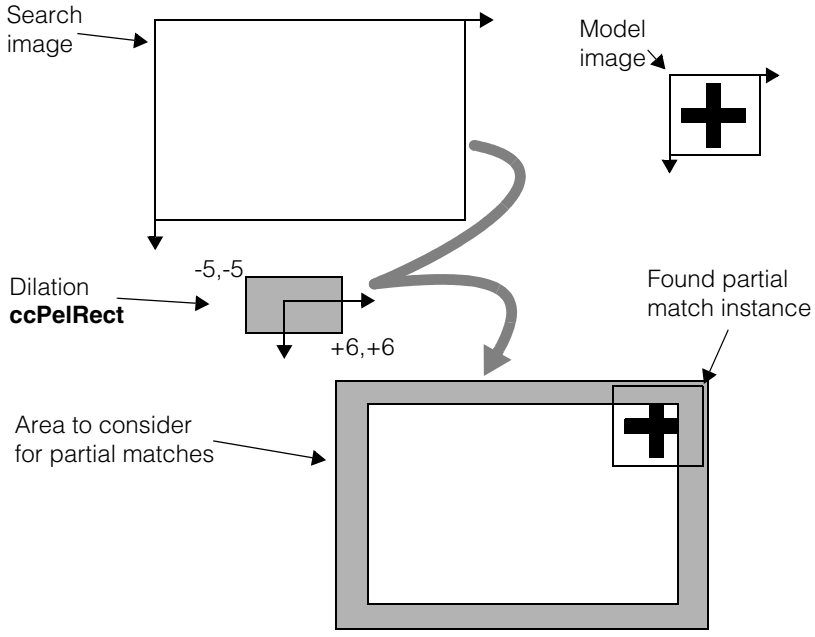
  ```
  ccPelRect(ccIPair(-left, -up), ccIPair(right+1, down+1))
  ```

where *left*, *right*, *up*, and *down* are the amount in pixels by which a model is allowed to lie outside that side of the search image.

For example, if you wanted to allow partial matches up to 5 pixels outside of the search image in any direction, you would supply the following **ccPelRect**:

```
ccPelRect(ccIPair(-5, -5), ccIPair(6, 6))
```

The following figure shows how the **ccPelRect** specifies the partial match search range.



If you specify a maximum search range of **ccPelRect(0, 0, 0, 0)**, CNLSearch will allow all partial matches (even as small as a single pixel). This special value (**ccPelRect(0, 0, 0, 0)**) is the default maximum partial match search range.

To specify that no partial matches are allowed, supply a value of **ccPelRect(0, 0, 1, 1)**.

Parameters

range The maximum partial match search range.

Throws

ccCnlSearchDefs::BadParams

range is not **ccPelRect(0, 0, 0, 0)** and *range* does not contain the point 0,0.

Notes

You can change the maximum partial match search range for a **ccCnlSearchModel** without needing to re-train the model.

Keep in mind that as the size of the partial match area increases, the potential for meaningless matches increases. For example, it is possible to specify a partial match range that will match a single pixel; such a match would provide no useful information. Also, as the size of the partial match area increases, search times will increase.

ccCnlSearchResult

```
#include <ch_cvl/cnlsrch.h>

class ccCnlSearchResult: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that holds a single CNLSearch search result.

Constructors/Destructors

ccCnlSearchResult

```
ccCnlSearchResult();
```

Constructs an empty **ccCnlSearchResult**.

Public Member Functions

found

```
bool found() const;
```

Returns *true* if this result received a score greater than or equal to the acceptance threshold specified for this search, *false* otherwise.

location

```
cc2Vect location() const;
```

Returns the location of the model origin in the client coordinate system of the search image.

score

```
double score() const;
```

Returns the result score (a number between 0 and 1). If **ccCnlSearchRunParams::weightScoreByOverlap()** is false, this value is equal to **ccCnlSearchResult::areaScore()**. **ccCnlSearchRunParams::weightScoreByOverlap()** is true, this value is equal to **ccCnlSearchResult::areaScore()*ccCnlSearchResult::areaCoverageScore()**.

edgeHit `c_UInt8 edgeHit() const;`

Returns a **c_UInt8** indicating whether the model was found against one or more edges of the search image. The return value is composed by ORing together one or more of the following values

ccCnlSearchResult::eNone
ccCnlSearchResult::eUnknownEdge
ccCnlSearchResult::eLeftEdge
ccCnlSearchResult::eTopEdge
ccCnlSearchResult::eRightEdge
ccCnlSearchResult::eBottomEdge

Notes

If you have specified *ccCnlSearchDefs::eAbsoluteSearch* or *ccCnlSearchDefs::eNormalizedSearch* for this search, **edgeHit()** will return either *ccCnlSearchResult::eNone* (if no edge was hit) or *ccCnlSearchResult::eUnknownEdge* (if any edge was it). Only searches performed using the *ccCnlSearchDefs::eNormalizedCnlpas* or *ccCnlSearchDefs::eNonlinearCnlpas* algorithms can report which edge was hit.

areaScore `double areaScore() const;`

Returns the area score (a number between 0 and 1). The area score represents the quality of the match regardless of the amount of model that appears in the image.

areaCoverageScore
`double areaCoverageScore() const;`

Returns the fraction of the model that lies within the search image. The returned value is in the range 0.0 through 1.0.

edgeScore `double edgeScore() const;`

Returns the edge score, a number between 0.0 and 1.0. The edge score is only valid for nonlinear mode searches.

Throws

ccCnlSearchDefs::NotComputed
The algorithm specified for this search was not *ccCnlSearchDefs::eNonlinearCnlpas*.

contrast `double contrast() const;`

Returns the ratio of the standard deviation of the pixels in the located instance of the model to the standard deviation of the care pixels in the model.

Throws

ccCnlSearchDefs::NotComputed
The algorithm specified for this search was *ccCnlSearchDefs::eNonlinearCnlpas*.

draw

```
void draw(ccGraphicList& graphList,
 c_UInt32 drawMode=ccCnlSearchDefs::eDrawStandard,
 const ccCvlString& label=ccCvlString()) const;
```

Appends result graphics for this **ccCnlSearchResult** to the supplied **ccGraphicList**.

Parameters

graphList The graphics list to append the graphics to.

drawMode The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccCnlSearchDefs::eDrawOrigin
ccCnlSearchDefs::eDrawLabel
ccCnlSearchDefs::eDrawBoundingBox
ccCnlSearchDefs::eDrawStandard

label If you include *ccCnlSearchDefs::eDrawLabel* in *drawMode*, then the contents of *label* are used to label the center of mass.

ccCnlSearchResult

ccCnlSearchResultSet

```
#include <ch_cvl/cnlsrch.h>

class ccCnlSearchResultSet: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that holds a list of **ccCnlSearchResult**s.

Constructors/Destructors

ccCnlSearchResultSet

```
ccCnlSearchResultSet();
```

The number of results and search time are initialized to zero.

Public Member Functions

results

```
const cmStd vector<ccCnlSearchResult>& results() const;
```

Return a vector containing a **ccCnlSearchResult** search result for each instance of the model found in the search image. If there are no results, the vector will have a size of zero.

time

```
double time() const;
```

Returns the execution time required to produce this result set (in seconds). This is the amount of time required to execute the **run()** member of **ccCnlSearchModel**.

algorithm

```
ccCnlSearchDefs::Algorithm algorithm() const;
```

Returns the algorithm used to generate this result set. Possible values are:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas

ccCnlSearchDefs::eNormalizedSearch
ccCnlSearchDefs::eAbsoluteSearch
ccCnlSearchDefs::kDefaultAlgorithm

For more information, see the description of *ccCnlSearchDefs* on page 973.

draw

```
void draw(ccGraphicList& graphList,
  c_UInt32 drawMode=ccCnlSearchDefs::eDrawStandard,
  bool drawOnlyFound = false) const;
```

Appends result graphics for this **ccCnlSearchResultSet** to the supplied **ccGraphicList**.

Parameters

graphList The graphics list to append the graphics to.

drawMode The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccCnlSearchDefs::eDrawOrigin
ccCnlSearchDefs::eDrawLabel
ccCnlSearchDefs::eDrawBoundingBox
ccCnlSearchDefs::eDrawStandard

drawOnlyFound If true, only found results are drawn. If false, all results are drawn.

ccCnlSearchRunParams

```
#include <ch_cvl/cnlsrch.h>

class ccCnlSearchRunParams: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that holds the parameters used to perform a search. This class is supplied as an argument to **ccCnlSearchModel::run()**.

Constructors/Destructors

ccCnlSearchRunParams

```
ccCnlSearchRunParams(
  ccCnlSearchDefs::Algorithm alg =
  ccCnlSearchDefs::kDefaultAlgorithm);
```

Constructs a **ccCnlSearchRunParams** using the given algorithm choice. The remaining members are assigned to the default values in **ccCnlSearchDefs**.

Parameters

alg — The algorithm to use in this search. *alg* must be one of the following:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas
ccCnlSearchDefs::eNormalizedSearch
ccCnlSearchDefs::eAbsoluteSearch
ccCnlSearchDefs::kDefaultAlgorithm

Public Member Functions

accuracy

```
ccCnlSearchDefs::Accuracy accuracy() const;
void accuracy(ccCnlSearchDefs::Accuracy acc);
```

- `ccCnlSearchDefs::Accuracy accuracy() const;`

  Returns the accuracy value specified in this **ccCnlSearchRunParams**. Returns one of the values listed below.

- `void accuracy(ccCnlSearchDefs::Accuracy acc);`

  Sets the accuracy for this **ccCnlSearchRunParams**. This determines the accuracy of the search. You must have trained the **ccCnlSearchModel** for the accuracy specified by *acc*.

  Parameters

  acc — The accuracy to set. *acc* must be one of the following values:

  ccCnlSearchDefs::eVeryFine
  ccCnlSearchDefs::eFine
  ccCnlSearchDefs::eCoarse
  ccCnlSearchDefs::kDefaultAccuracy

  Notes

  The **run()** member of **ccCnlSearchModel** may search at a higher accuracy than you specify with this function if the model image is not suitable for low-accuracy searching.

algorithm

```
ccCnlSearchDefs::Algorithm algorithm() const;
void algorithm(ccCnlSearchDefs::Algorithm alg);
```

- `ccCnlSearchDefs::Algorithm algorithm() const;`

  Returns the algorithm specified in this **ccCnlSearchRunParams**.

- `void algorithm(ccCnlSearchDefs::Algorithm alg);`

  Sets the algorithm for this **ccCnlSearchRunParams**.

Parameters

alg The algorithm for this search. *alg* must be one of the following values:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas
ccCnlSearchDefs::eNormalizedSearch
ccCnlSearchDefs::eAbsoluteSearch
ccCnlSearchDefs::kDefaultAlgorithm

For more information, see the description of *ccCnlSearchDefs* on page 973.

Notes

A model can only be searched for using an algorithm it was trained for.

accept

```
double accept() const;
void accept(double thresh);
```

- `double accept() const;`

Returns the acceptance threshold specified in this **ccCnlSearchRunParams**.

- `void accept(double thresh);`

Sets the acceptance threshold for this **ccCnlSearchRunParams**. The acceptance threshold must be between 0.0 and 1.0 inclusive. Set the acceptance threshold to be below the lowest score that an actual instance of the model is expected to receive.

Parameters

thresh The acceptance threshold to use.

Throws

ccCnlSearchDefs::BadParams
If *thresh* is not between 0.0 and 1.0.

confusion

```
double confusion() const;
void confusion(double thresh);
```

- `double confusion() const;`

Returns the confusion threshold specified in this **ccCnlSearchRunParams**.

- `void confusion(double thresh);`

  Sets the confusion threshold for this **ccCnlSearchRunParams**. The confusion threshold must be between 0.0 and 1.0 inclusive. Set the confusion threshold to be equal to the highest score that a feature in the model that is *not* an instance of the model is expected to receive.

  Parameters

  thresh The confusion threshold to use.

  Notes

  The tool uses the confusion threshold as a hint regarding how confusing the search image is likely to be. A high value for confusion slows the search but ensures that wrong features are not found in a confusing scene. A low confusion threshold increases the search speed for scenes that are not confusing.

  The confusion threshold must be greater than or equal to the acceptance threshold.

  Throws

  ccCnlSearchDefs::BadParams
  If *thresh* is not between 0.0 and 1.0.

maxNumResults

```
c_Int16 maxNumResults() const;
void maxNumResults(c_Int16 num);
```

- `c_Int16 maxNumResults() const;`

  Returns the maximum number of results specified in this **ccCnlSearchRunParams**.

- `void maxNumResults(c_Int16 num);`

  Sets the maximum number of results for this **ccCnlSearchRunParams**. The **run()** member of **ccCnlSearchModel** will return up to this number of search results.

  Parameters

  num The maximum number of results.

highThreshold

```
c_UInt8 highThreshold() const;
void highThreshold(c_UInt8 hiThresh);
```

- `c_UInt8 highThreshold() const;`

  Returns the edge detection high threshold value specified in this **ccCnlSearchRunParams**.

- `void highThreshold(c_UInt8 hiThresh);`

  Sets the edge detection high threshold value for **ccCnlSearchRunParams**. All edges with strength above this threshold are included in the edge map of the search image.

  Parameters

  hiThresh The high threshold value.

  Notes

  This parameter is only used when the search algorithm is *ccCnlSearchDefs::eNonlinearCnlpas*. The high threshold must be greater than the low threshold.

lowThreshold

```
c_UInt8 lowThreshold() const;
void lowThreshold(c_UInt8 loThresh);
```

- `c_UInt8 lowThreshold() const;`

  Returns the edge detection low threshold value specified in this **ccCnlSearchRunParams**.

- `void lowThreshold(c_UInt8 loThresh);`

  Sets the edge detection low threshold value for this **ccCnlSearchRunParams**. Edges with strengths below this threshold are excluded from the edge map. Edges with strengths between the low threshold and the high threshold are included in the edge map if they are 8-connected to an edge above the high threshold, either directly or through one or more other edges above the low threshold but below the high threshold.

  Parameters

  loThresh The low threshold value.

  Notes

  This parameter is only used when the search algorithm is *ccCnlSearchDefs::eNonlinearCnlpas*. The low threshold must be less than the high threshold.

xyOverlap

```
double xyOverlap() const;
void xyOverlap(double overlapThresh);
```

- `double xyOverlap() const;`

  Returns the maximum allowed percentage of overlap between two search results.

- `void xyOverlap(double overlapThresh);`

  Sets the maximum allowed percentage of overlap between two search results. If more than the specified percentage of the results' areas are overlapped, then the result with the lower score is discarded. If the percentage of overlap is below the threshold, then both results are returned.

  Parameters

  overlapThresh The overlap percentage threshold, specified in the range 0.0 through 1.0. Specify a value of 1.0 (the default) to obtain the behavior seen in earlier CNLSearch releases.

  Throws

  ccCnlSearchDefs::BadParams
  overlapThresh is not in the range 0.0 through 1.0.

weightScoreByOverlap

```
bool weightScoreByOverlap() const;

void weightScoreByOverlap(bool);
```

- `bool weightScoreByOverlap() const;`

  This function returns true if CNLSearch weights the scores of found model instances by the percentage of the model that appears within the search window, false otherwise.

  Notes

  This value is only meaningful if you are using partial match searching.

  Note that this function is not related to the **xyOverlap()** function.

- `void weightScoreByOverlap(bool weight);`

  This function controls whether CNLSearch weights the scores of found model instances by the percentage of the model that appears within the search window.

  Parameters

  weight If true, CNLSearch will multiply model scores by the percentage of the model that lies within the search image. If false, scores will not be weighted.

Notes

This value is only meaningful if you are using partial match searching.

Note that this function is not related to the **xyOverlap()** function.

Regardless of **ccCnlSearchRunParams::weightScoreByOverlap()**, **ccCnlSearchResult::areaScore()** will return the quality of the match of whatever part of the model in the image.

Regardless of **ccCnlSearchRunParams::weightScoreByOverlap()**, **ccCnlSearchResult::areaCoverageScore()** will return the fraction of the area of the model that is in the image.

If **ccCnlSearchRunParams::weightScoreByOverlap()** is true, then **ccCnlSearchResult::score()** is equal to **areaScore() * areaCoverageScore()**. If **ccCnlSearchRunParams::weightScoreByOverlap()** is false, then **ccCnlSearchResult::score()** is equal to **areaScore()**.

ccCnlSearchRunParams

ccCnlSearchTrainParams

```
#include <ch_cvl/cnlsrch.h>

class ccCnlSearchTrainParams: public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that holds the parameters used to train a CNLSearch model. This class is supplied as an argument to the **train()** member of the **ccCnlSearchModel** class.

Constructors/Destructors

ccCnlSearchTrainParams

```
ccCnlSearchTrainParams();

ccCnlSearchTrainParams(c_UInt16 algs);
```

- `ccCnlSearchTrainParams();`

  Constructs a **ccCnlSearchTrainParams** using the default values in **ccCnlSearchDefs**.

- `ccCnlSearchTrainParams(c_UInt16 algs);`

  Constructs a **ccCnlSearchTrainParams** using the given algorithm choices. The remaining members are assigned to the default values in **ccCnlSearchDefs**.

  Parameters

  algs — The algorithms for which to train this model. *algs* must be formed by ORing together any of the following values:

  ccCnlSearchDefs::eNormalizedCnlpas
  ccCnlSearchDefs::eAbsoluteCnlpas
  ccCnlSearchDefs::eNonlinearCnlpas
  ccCnlSearchDefs::eNormalizedSearch
  ccCnlSearchDefs::eAbsoluteSearch
  ccCnlSearchDefs::kDefaultAlgorithm

For more information, see the description of *ccCnlSearchDefs* on page 973.

Public Member Functions

accuracies

```
c_UInt16 accuracies() const;

void accuracies(c_UInt16 accs);
```

- `c_UInt16 accuracies() const;`

  Returns a **c_UInt16** that is formed by ORing together all of the accuracies for which this **ccCnlSearchTrainParams** is trained. The return value is formed by ORing together one or more of the following values:

  ccCnlSearchDefs::eVeryFine
  ccCnlSearchDefs::eFine
  ccCnlSearchDefs::eCoarse

- `void accuracies(c_UInt16 accs);`

  Sets the accuracies for which this **ccCnlSearchTrainParams** is to be trained. You can only search using an accuracy which was specified at training.

  Parameters

  accs — The accuracies to set. *acc* must be formed by ORing together one or more of the following values:

  ccCnlSearchDefs::eVeryFine
  ccCnlSearchDefs::eFine
  ccCnlSearchDefs::eCoarse

  Notes

  The **train()** member of **ccCnlSearchModel** may train the model at a higher accuracy than you specify with this function if the model image is not suitable for low-accuracy searching.

  In most cases, you should specify *ccCnlSearchDefs::eVeryFine*.

accuracy

```
ccCnlSearchDefs::Accuracy accuracy() const;
void accuracy(ccCnlSearchDefs::Accuracy acc);
```

- `ccCnlSearchDefs::Accuracy accuracy() const;`

  Return the accuracy value specified in this **ccCnlSearchTrainParams**. Returns one of the values listed below.

  ccCnlSearchDefs::eVeryFine
  ccCnlSearchDefs::eFine
  ccCnlSearchDefs::eCoarse

- `void accuracy(ccCnlSearchDefs::Accuracy acc);`

  Sets the accuracy for this **ccCnlSearchTrainParams**. The accuracy you set determines the accuracy with which you can search at run time.

  Parameters

  acc The accuracy to set. *acc* must be one of the following values:

  ccCnlSearchDefs::eVeryFine
  ccCnlSearchDefs::eFine
  ccCnlSearchDefs::eCoarse
  ccCnlSearchDefs::kDefaultAccuracy

  Notes

  The **train()** member of **ccCnlSearchModel** may train the model at a higher accuracy than you specify with this function if the model image is not suitable for low-accuracy searching.

  In most cases, you should specify *ccCnlSearchDefs::eVeryFine*.

algorithms

```
c_UInt16 algorithms() const;
void algorithms(c_UInt16 alg);
```

- `c_UInt16 algorithms() const;`

  Returns the algorithm specified in this **ccCnlSearchTrainParams**.

- `void algorithms(c_UInt16 alg);`

  Sets the algorithm for this **ccCnlSearchTrainParams**.

Parameters

alg The algorithm for which to train the model. *alg* must be formed by ORing together any of the following values:

ccCnlSearchDefs::eNormalizedCnlpas
ccCnlSearchDefs::eAbsoluteCnlpas
ccCnlSearchDefs::eNonlinearCnlpas
ccCnlSearchDefs::eNormalizedSearch
ccCnlSearchDefs::eAbsoluteSearch
ccCnlSearchDefs::kDefaultAlgorithm

For more information, see the description of *ccCnlSearchDefs* on page 973.

Notes

When you perform the search, you can specify only an algorithm that is among those that you specify when you train the model.

Throws

ccCnlSearchDefs::BadParams
algs is not the result of ORing together the values in *ccCnlSearchDefs::Algorithm*.

ccCnlSearchDefs::NotImplemented
algs includes *ccCnlSearchDefs*::*eAbsoluteCnlpas*.

highThreshold

```
c_UInt8 highThreshold() const;}

void highThreshold(c_UInt8 highThresh);
```

- `c_UInt8 highThreshold() const;`

  Returns the edge detection high threshold value specified in this **ccCnlSearchTrainParams**.

- `void highThreshold(c_UInt8 highThresh);`

  Sets the edge detection high threshold value for this **ccCnlSearchTrainParams**. All edges with strengths above this threshold are included in the edge map of the model image.

  Parameters

  hiThresh The high threshold value.

Notes

This parameter is only used when the search algorithm is *ccCnlSearchDefs::eNonlinearCnlpas*. The high threshold must be greater than the low threshold.

lowThreshold

```
c_UInt8 lowThreshold() const;

void lowThreshold(c_UInt8 lowThresh);
```

- `c_UInt8 lowThreshold() const;`

  Returns the edge detection low threshold value specified in this **ccCnlSearchTrainParams**.

- `void lowThreshold(c_UInt8 lowThresh);`

  Sets the edge detection low threshold value for this **ccCnlSearchTrainParams**. Edges with strengths below this threshold are excluded from the edge map. Edges with strengths between the low threshold and the high threshold are included in the edge map if they are 8-connected to an edge above the high threshold, either directly or through one or more other edges above the low threshold but below the high threshold.

  Parameters

  lowThresh The low threshold value.

  Notes

  This parameter is only used when the search algorithm is *ccCnlSearchDefs::eNonlinearCnlpas*. The low threshold must be less than the high threshold.

useAdvancedTrainingSearch

```
void useAdvancedTrainingSearch(bool u);

bool useAdvancedTrainingSearch() const;
```

- `void useAdvancedTrainingSearch(bool u);`

  If *u* is true, CNLSearch models trained using this **ccCnlSearchTrainParams** object will use advanced training; if *u* is false, they will use standard model training.

  Parameters

  u Boolean parameter: if true use advanced training, if false use standard training

- `bool useAdvancedTrainingSearch() const;`

  Returns true if CNLSearch models trained using this **ccCnlSearchTrainParams** object will use advanced training, false if they will use standard model training.

  A default-constructed **ccCnlSearchTrainParams** will not use advanced training; you must explicitly enable it by using **useAdvancedTrainingSearch()**.

ccColor

```
#include <ch_cvl/color.h>

class ccColor;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class is used to specify colors. CVL allows you to describe colors two ways: as an RGB color composed of red, green, and blue components, or as an indexed color whose actual color depends on its index within a color table.

CVL provides sixteen stock colors that you can use in your application.

Constructors/Destructors

ccColor

```
ccColor(const ccRGB& color);
ccColor(c_UInt8 index = 0);
ccColor(c_UInt8 r, c_UInt8 g, c_UInt8 b);
```

- `ccColor(const ccRGB& color);`

  Creates a new RGB color.

  Parameters

  color The color object that specifies the RGB color.

- `ccColor(c_UInt8 index = 0);`

  Creates a new indexed color.

  Parameters

  index The index in the color palette used to specify this color.

- `ccColor(c_UInt8 r, c_UInt8 g, c_UInt8 b);`

  Creates a new RGB color.

Parameters

r	The red component. Must be a value from 0 to 255.
g	The green component. Must be a value from 0 to 255.
b	The blue component. Must be a value from 0 to 255.

Enumerations

These enumerated values specify the type of color.

Value	Meaning
eStockColor = 0x1	A CVL stock color
eRGBColor = 0x2	An RGB color
eIndexColor = 0x4	An index color

Operators

operator==

```
bool operator==(const ccColor& other) const;
```

Returns true if this color is equal to another color.

Parameters

other	The other color to compare to this color.

operator!=

```
bool operator!=(const ccColor& other) const;
```

Returns true if this color is not equal to another color.

Parameters

other	The other color to compare to this color.

Public Member Functions

isStockColor

```
bool isStockColor() const;
```

Returns true if this is one of the stock colors defined on page 1024. If this function returns true, then **isIndexColor()** and **isRGBColor()** will also return true.

isIndexColor `bool isIndexColor() const;`

Returns true if this color is an indexed color.

isRGBColor `bool isRGBColor() const;`

Returns true if this color is an RGB color.

Notes

Base class RGB getters return true only if this method is true.

index `c_UInt8 index() const;`

Returns the index of this color within the color map.

Notes

The result is valid only if the color is an indexed color such that **ccColor::isIndexColor()** is true. If not, **ccColor::index()** returns the value 0.

rgb `c_UInt32 rgb() const;`

Returns this color's RGB value as a single 32-bit value in the format `0x00rrggbb`.

Notes

The result is valid only if **isRGBColor()** is true. If it is not, the application terminates with an error message.

bgr `c_UInt32 bgr() const;`

Returns this color's RGB value as a single 32-bit value in the format `0x00bbggrr`.

Notes

The result is valid only if **isRGBColor()** is true. If it is not, the application terminates with an error message.

rgb15 `c_UInt16 rgb15() const;`

Returns this color's RGB value as a single "5-5-5" 16-bit value, using 5 bits for each component. The low-order 3 bits of each original component are ignored.

Notes

The result is valid only if **isRGBColor()** is true. If it is not, the application terminates with an error message.

rgb16 `c_UInt16 rgb16() const;`

Returns this color's RGB values as a single "5-6-5" 16-bit value, using 5 bits for the red component, 6 bits for this green component, and 5 bits for the blue component. The low-order bits of the original red and blue components and the low-order 2 bits of the green component are ignored.

Notes

The result is valid only if **isRGBColor()** is true. If it is not, the application terminates with an error message.

rgbStruct `const ccRGB& rgbStruct() const;`

Returns the **ccRGB** structure for this color.

Notes

The result is valid only if **isRGBColor()** is true. If it is not, the application terminates with an error message.

Static Functions

The following static functions return stock RGB colors that are available at static initialization time. Stock colors are defined as both indexed colors and RGB colors: **isRGBColor()** and **isIndexColor()** return true for these colors.

blackColor `static const ccColor& blackColor();`

dkGreyColor `static const ccColor& dkGreyColor();`

greyColor `static const ccColor& greyColor();`

ltGreyColor `static const ccColor& ltGreyColor();`

whiteColor `static const ccColor& whiteColor();`

redColor `static const ccColor& redColor();`

dkRedColor `static const ccColor& dkRedColor();`

orangeColor `static const ccColor& orangeColor();`

yellowColor `static const ccColor& yellowColor();`

greenColor `static const ccColor& greenColor();`

dkGreenColor `static const ccColor& dkGreenColor();`

cyanColor `static const ccColor& cyanColor();`

blueColor `static const ccColor& blueColor();`

purpleColor `static const ccColor& purpleColor();`

magentaColor `static const ccColor& magentaColor();`

passColor `static const ccColor& passColor();`

ccColorMatchDefs

```
#include <ch_cvl/colmatch.h>

class ccColorMatchDefs
```

Enumerations

ColorMatchColorDistanceMetric

```
enum ColorMatchColorDistanceMetric
```

This enumeration defines constants for the Color Match tool distance metric.

Value	Meaning
eWeightedEuclideanDistance = 0x1	The square root of the weighted sum of squared distance in each color plane. $\sqrt{\sum_{i=0,1,2} w_i(c1_i - c2_i)^2}$
kDefaultColorDistanceMetric = *eWeightedEuclideanDistance*	Default

ccColorMatchDefs

ccColorMatchResult

```
#include <ch_cvl/colmatch.h>

class ccColorMatchResult
```

Class Properties

Copyable	Yesy
Derivable	Yes
Archiveable	Simple

This class contains the results obtained by calling **cfColorMatch()** or **ccColorMatchTool::run()**.

Constructors/Destructors

ccColorMatchResult

```
ccColorMatchResult();
```

Creates a default ccColorMatchResult object with the following defaults:

Parameter	Default value
label	0
maxScore	0
confidenceScore	0
matchScores	empty vector
colorDistances	empty vector

Public Member Functions

maxScore

```
double maxScore() const;
```

Returns the maximum match score for al the color references in the color classification.The score ranges from 0.0 to 1.0, where 1.0 is a perfect match.

Notes

If there is only one reference color, the result is the match score for the reference color.

ccColorMatchResult

confidenceScore

```
double confidenceScore() const;
```

Return the confidence score of the color classification. The score ranges from 0.0 to 1.0, where 1.0 is the highest confidence.

Notes

Confidence is computed as:

$$\frac{\text{ghest score} - \text{second highest scor}}{\text{ghest score} + \text{second highest scor}}$$

label

```
c_Int32 label() const;
```

Returns the zero-based index that specifies the position of the color with the highest score in the color collection.

numberOfReferenceColors

```
c_Int32 numberOfReferenceColors() const;
```

Returns the number of color references used for color match.

matchScores

```
const cmStd vector<double>& matchScores() const;
```

Return the vector of match scores for all the color references.

ccColorMatchRunParams

```
#include <ch_cvl/colmatch.h>

class ccColorMatchRunParams
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class defines the parameters used by the Color Match tool: the color space used, the distance metric used to calculate the color distance, and the weights for each color component.

See **cfColorMatch()**.

Constructors/Destructors

ccColorMatchRunParams

```
ccColorMatchRunParams();
```

Creates a ccColorMatchRunParams object with the following default values:

Parameter	Default value
colorSpace	**ccColorSpaceDefs::kDefaultColorSpace**
distanceMetricType	**ccColorMatchDefs::eWeightedEuclideanDistance**
weights	(1.0, 1.0, 1.0)

Public Member Functions

distanceMetricType

```
ccColorMatchDefs::ColorMatchColorDistanceMetric
   distanceMetricType() const;

void distanceMetricType(
   ccColorMatchDefs::ColorMatchColorDistanceMetric
         distanceMetricType);
```

- `ccColorMatchDefs::ColorMatchColorDistanceMetric distanceMetricType() const;`

  Returns the distance metric used in color distance calculation.

- `void distanceMetricType( ccColorMatchDefs::ColorMatchColorDistanceMetric distanceMetricType);`

  Sets the distance metric used in color distance calculation.

  Parameters

  distanceMetricType
  The distance metric to use.

  Throws

  ccColorMatchDefs::BadParams
  distanceMetricType is not defined in **ccColorMatchDefs::ColorMatchColorDistanceMetric**.

colorSpace

```
ccColorSpaceDefs::ColorSpace colorSpace() const;

void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);
```

- `ccColorSpaceDefs::ColorSpace colorSpace() const;`

  Returns the color space used to perform color distance calculations.

- `void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);`

  Sets the color space used to perform color distance calculations.

  Parameters

  colorSpace The color space used: RGB or HSI.

Throws

ccColorMatchDefs::BadColorSpace
colorSpace is neither **ccColorSpaceDefs::eRGB** nor **ccColorSpaceDefs::eHSI**

weights

```
cc3Vect weights() const;
void weights(const cc3Vect& newValue);
```

- `cc3Vect weights() const;`

  Returns the weights used in calculating color distance for each color component.

- `void weights(const cc3Vect& newValue);`

  Sets the weights used in calculating color distance for each color component.

  Parameters

  newValue The weights.

  Throws

  ccColorMatchDefs::BadParams
  an element of *newValue* < 0
  newValue is (0,0,0)

ccColorMatchRunParams

ccColorRange

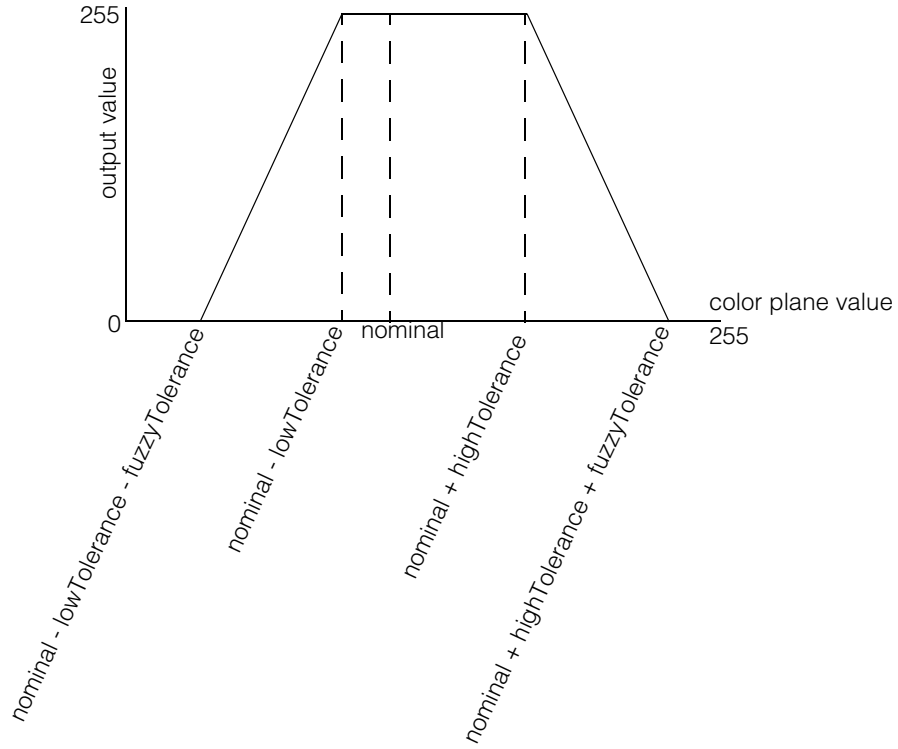
```
#include <ch_cvl/coltool.h>

class ccColorRange
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a color range that defines a nominal color value and high and low tolerance values.



Constructors/Destructors

ccColorRange `ccColorRange();`

Creates a color range object with the following default values:

Parameter	Default value
nominal	default **ccColorValue**
lowTolerance	(0,0.0)
highTolerance	(0,0,0)
fuzzyTolerance	(0,0,0)

Public Member Functions

nominal

```
ccColorValue nominal() const;
void nominal(const ccColorValue& newColor);
```

- `ccColorValue nominal() const;`

  Returns the nominal color value for this color range.

- `void nominal(const ccColorValue& newColor);`

  Sets the nominal color value for this color range.

  Parameters
  newColor The nominal color value.

lowTolerance

```
cc3Vect lowTolerance() const;
void lowTolerance(const cc3Vect& newValue);
```

- `cc3Vect lowTolerance() const;`

  Returns the low tolerance for this color range.

- `void lowTolerance(const cc3Vect& newValue);`

  Sets the low tolerance for this color range.

Parameters
newValue The low tolerance for this color range.

Throws
ccColorToolDefs::BadColorValue
one or more of the values in newValue is less than zero.

highTolerance

```
cc3Vect highTolerance() const;
void highTolerance(const cc3Vect& newValue);
```

- `cc3Vect highTolerance() const;`

  Returns the high tolerance for this color range.

- `void highTolerance(const cc3Vect& newValue);`

  Sets the high tolerance for this color range.

  Parameters
  newValue The high tolerance for this color range.

  Throws
  ccColorToolDefs::BadColorValue
  one or more of the values in newValue is less than zero.

fuzzyTolerance

```
cc3Vect fuzzyTolerance() const;
void fuzzyTolerance(const cc3Vect& newValue);
```

- `cc3Vect fuzzyTolerance() const;`

  Returns the fuzzy tolerance for this range.

- `void fuzzyTolerance(const cc3Vect& newValue);`

  Sets the fuzzy tolerance for this color range. Fuzzy tolerance is the same on both side of the nominal value.

  Parameters
  newValue The fuzzy tolerance for this color range.

  Throws
  ccColorToolDefs::BadColorValue
  one or more of the values in newValue is less than zero.

ccColorRange

ccColorSpaceDefs

```
#include <ch_cvl/colspace.h>
```

```
class ccColorSpaceDefs
```

A name space that decribes how a 3-plane color space is interpreted.

Enumerations

ColorSpace

```
enum ColorSpace
```

This enumeration defines how the values of a 3-plane image are interpreted.

Value	Meaning
eRGB = 0x1	Pixel values refer to RGB color space.
eHSI = 0x2	Pixel values refer to HSI color space.
eUnknown = 0x3	The color space is not known.

ccColorSpaceDefs

ccColorStatisticsParams

```
#include <ch_cvl/coltool.h>

class ccColorStatisticsParams
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class defines the values needed to compute color statistics from a three-plane color image. This class is used by

- **cfGetColorRangeFromImage()**
- **cfGetColorRangeFromImageRegion()**
- **cfGetColorStatisticsFromImage()**
- **cfGetColorStatisticsFromImageRegion()**
- **cfGetSimpleColorFromImage()**
- **cfGetSimpleColorFromImageRegion()**

Constructors/Destructors

ccColorStatisticsParams

```
ccColorStatisticsParams();

ccColorStatisticsParams(double minIntensity,
                        double minSaturation);
```

```
ccColorStatisticsParams();
```

Creates a default **ccColorStatisticsParams** with the following default values:

Parameter	Default value
minIntensity()	0
minSaturation()	0

```
ccColorStatisticsParams(double minIntensity,
                        double minSaturation);
```

Creates a **ccColorStatisticsParams** with the specified *minIntensity* and *minSaturationvalues*.

Parameters

minIntensity The minimum intensity value.

minSaturation The minimum saturation value.

Throws

ccColorToolDefs::BadParams
at least one parameter is less than zero.

Public Member Functions

minIntensity

```
double minIntensity() const;

void minIntensity(double newValue);
```

```
double minIntensity() const;
```

Returns the minimum intensity.

```
void minIntensity(double newValue);
```

Sets the minimum intensity.

Parameters

newValue The new minimum intensity.

Throws

ccColorToolDefs::BadParams
newvalue is less than zero.

minSaturation

```
double minSaturation() const;

void minSaturation(double newValue);
```

```
double minSaturation() const;
```

Returns the minimum saturation.

```
void minSaturation(double newValue);
```

Sets the minimum saturation.

Parameters

newValue The new minimum saturation.

Throws

ccColorToolDefs::BadParams
newvalue is less than zero.

ccColorStatisticsParams

ccColorStatisticsResult

```
#include <ch_cvl/coltool.h>

class ccColorStatisticsResult
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class is used by **cfGetColorStatisticsFromImage()** and **cfGetColorStatisticsFromImageRegion()** to return the mean color and standard deviations of an image.

Constructors/Destructors

ccColorStatisticsResult

```
ccColorStatisticsResult(){};
```

Creates a ccColorStatisticsResult object with the mean color set the the default XX and the standard deviation vector set to (0,0,0).

Public Member Functions

meanColor

```
ccColorValue meanColor() const { return meanColor_;}
```

Returns the mean color.

standardDeviation

```
cc3Vect standardDeviation() cons;
```

Returns the vector of standard deviations.

ccColorStatisticsResult

ccColorValue

```
#include <ch_cvl/coltool.h>

class ccColorValue
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes the value of a color as a triplet and specifies the color space of those values, typically RGB or HSI.

Constructors/Destructors

ccColorValue

```
ccColorValue();
```

Creates a color value using whose color values is (0,0,0) and whose color space is **ccColorSpaceDefs::kDefaultColorSpace**.

Public Member Functions

colorSpace

```
ccColorSpaceDefs::ColorSpace colorSpace() const;

void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);
```

- ```
  ccColorSpaceDefs::ColorSpace colorSpace() const;
  ```

  Returns the color space of the color value.

- ```
  void colorSpace(ccColorSpaceDefs::ColorSpace colorSpace);
  ```

  Sets the color space of the color value.

  Parameters

  colorSpace The color space.

  Throws

  ccColorToolDefs::BadColorSpace
  colorSpace is not defined in **ccColorSpaceDefs::ColorSpace**.

ccColorValue

colorValues

```
cc3Vect colorValues() const;
void colorValues(const cc3Vect& newValue);
```

- `cc3Vect colorValues() const;`

  Returns the three plane values that correspond to this color.

- `void colorValues(const cc3Vect& newValue);`

  Sets the three plane values that correspond for this color.

  Parameters

  newValue The 3-plane values

  Throws

  ccColorToolDefs::BadColorValue
  any of the values in *newValue* is less than zero

ccCompleteCallbackProp

```
#include <ch_cvl/prop.h>

class ccCompleteCallbackProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

As of CVL 6.1, this class is one of two ways to register a callback class for the complete state:

Method	Features	See
ccAcqFifo:: completeInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	*completeInfoCallback* on page 232
ccCompleteCallbackProp	Calls your callback function.	this section

If you are writing a new CVL application, consider using **ccAcqFifo::completeInfoCallback()** to register your completion callback function.

This class describes the completion callback property of an acquisition FIFO queue. The class contains a pointer to a function you write that is called by the acquisition engine when acquisition is complete.

The callback function you write should set flags or semaphores in your application that allow your program to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition software, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

An acquisition enters the complete state (**ccAcqFifo::isComplete()** returns true) when the next invocation of **ccAcqFifo::baseComplete()** would return the acquired image (if the acquisition was successful) or an unbound pel buffer (if the acquisition was not successful).

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccCompleteCallbackProp

```
ccCompleteCallbackProp();

ccCompleteCallbackProp(const ccCallbackPtrh& callback);
```

- `ccCompleteCallbackProp();`

  Creates a new acquisition complete callback property not associated with any FIFO. The default callback function is an unbound handle, meaning no callback function is invoked.

- `ccCompleteCallbackProp(const ccCallbackPtrh& callback);`

  Creates a new acquisition complete callback property not associated with any FIFO, registering the function pointed to by *callback* as the completion callback function.

  Parameters

  callback — Effectively, the callback function to be invoked when the acquisition process enters the complete state. This is generally when the next invocation of **ccAcqFifo::baseComplete()** would return the acquired image (if the acquisition is successful). An unbound handle means no callback will occur. Technically, *callback* is a pointer handle to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

  Notes

  The acquisition complete callback function is invoked whether the acquisition completed successfully or not.

Public Member Functions

completeCallback

```
void completeCallback(const ccCallbackPtrh& callback);

const ccCallbackPtrh& completeCallback() const;
```

- `void completeCallback(const ccCallbackPtrh& callback);`

  callback points to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

  Parameters

callback	Effectively, the callback function to be invoked when the acquisition process enters the complete state. This is generally when the next invocation of **ccAcqFifo::baseComplete()** would return the acquired image (if the acquisition is successful). An unbound handle means no callback will occur. Technically, *callback* is a pointer handle to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

  Notes

  The acquisition complete callback function is invoked whether the acquisition completed successfully or not.

- `const ccCallbackPtrh& completeCallback() const;`

  Returns the callback function to be invoked when the acquisition enters the completed state.

ccCompleteCallbackProp

ccCompositeColorMatchRunParams

```
#include <ch_cvl/colmatch.h>

class ccCompositeColorMatchRunParams
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes the parameters used to run the Composite Color Match tool.

Constructors/Destructors

ccCompositeColorMatchRunParams

```
ccCompositeColorMatchRunParams();
```

Creates a **ccCompositeColorMatchRunParams** with the following default values:

Parameter	Default value
normalizeIntensity	False
matchingAccuracy	1.0

Public Member Functions

normalizeIntensity

```
bool normalizeIntensity() const;

void normalizeIntensity(bool newValue);
```

- `bool normalizeIntensity() const;`

  Returns whether the Composite Color Match tool should normalize the image intensity.

- `void normalizeIntensity(bool newValue);`

  Sets whether the Composite Color Match tool should normalize the image intensity.

Parameters

newValue True to normalize intensity, False otherwise. The default is False.

matchingAccuracy

```
double matchingAccuracy() const;

void matchingAccuracy(double newValue);
```

- `double matchingAccuracy() const;`

  Returns the matching accuracy,

- `void matchingAccuracy(double newValue);`

  Sets the matching accuracy,

  Parameters

  newValue The matching accuracy.

  Throws

  ccColorMatchDefs::BadParams
  newValue > 1 or <= 0

ccCompositeColorMatchTool

```
#include <ch_cvl/colmatch.h>

class ccCompositeColorMatchTool;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Use this tool for composite color matching. For simple color matching, use **cfColorMatch()**.

The composite color tool compares the color content of images, Images with the same or similar color distribution (similar color content) are considered with good match.

Constructors/Destructors

ccCompositeColorMatchTool

```
ccCompositeColorMatchTool();
```

Creates an untrained **ccCompositeColorMatchTool** object

Public Member Functions

train

```
void train(
   const cmStd vector<cc3PlanePelBuffer>& colorImages,
   const ccCompositeColorMatchTrainParams& trainParams);

void train(
   const cmStd vector<cc3PlanePelBuffer>& colorImages,
   const cmStd vector<ccShapePtrh> & regions,
   const ccCompositeColorMatchTrainParams& trainParams);

void train(
   const cmStd vector<cc3PlanePelBuffer>& colorImages,
   const cmStd vector<ccPelBuffer_const<c_UInt8> >& masks,
   const ccCompositeColorMatchTrainParams& trainParams);

void train(
   const cc3PlanePelBuffer& colorImage,
   const ccCompositeColorMatchTrainParams& trainParams);
```

- ```
  void train(
     const cmStd vector<cc3PlanePelBuffer>& colorImages,
     const ccCompositeColorMatchTrainParams& trainParams);
  ```

  Trains the *colorImages* using the *trainParams*. If the tool is already trained, it is untrained and retrained.

  Parameters

  colorImages The images used to train.

  trainParams A **ccCompositeColorMatchTrainParams** object that specifies the sampling percentage and the size of the Gaussian smoothing kernel used to train the images.

  Throws

  ccColorMatchDefs::BadParams
  colorImages is empty.

  ccColorMatchDefs::BadImage
  colorImages contains an unbound image.

  ccColorMatchDefs::NotEnoughSamples
  Not enough pels (< 1) in the image.

  ccColorMatchDefs::BadColorSpace
  colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.
  ```

- ```
  void train(
     const cmStd vector<cc3PlanePelBuffer>& colorImages,
     const cmStd vector<ccShapePtrh> & regions,
     const ccCompositeColorMatchTrainParams& trainParams);
  ```

  Trains the portion of *colorImages* that intersect the corresponding region using the *trainParams*. If the tool is already trained, it is untrained and retrained.

  Parameters

  colorImages The images used to train.

  regions A vector of shapes that defines a region for each image. If a region is NULL, the whole image is used.

  trainParams A **ccCompositeColorMatchTrainParams** object that specifies the sampling percentage and the size of the Gaussian smoothing kernel used to train the images.

  Throws

  ccColorMatchDefs::BadParams
  colorImages is empty.

  ccColorMatchDefs::BadParams
  The number of images in *colorImages* does not match the number of regions in *regions*.

  ccColorMatchDefs::BadImage
  colorImages contains an unbound image.

  ccColorMatchDefs::NotEnoughSamples
  Not enough pels (< 1) in the image.

  ccColorMatchDefs::BadColorSpace
  colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.

  ccColorMatchDefs::ShapeIsNotRegion
  one of the regions in *regions* is not a region shape.

- ```
  void train(
     const cmStd vector<cc3PlanePelBuffer>& colorImages,
     const cmStd vector<ccPelBuffer_const<c_UInt8> >& masks,
     const ccCompositeColorMatchTrainParams& trainParams);
  ```

  Trains the portion of *colorImages* that are described by the corresponding mask using the *trainParams*. If the tool is already trained, it is untrained and retrained.

Parameters

colorImages The images used to train.

masks A vector of image masks. Pels whose value is 0 are don't care pixels; pels whose values are 255 are care pixels. All other values throw an error.

trainParams A **ccCompositeColorMatchTrainParams** object that specifies the sampling percentage and the size of the Gaussian smoothing kernel used to train the images.

Throws

ccColorMatchDefs::BadParams
colorImages is empty.

ccColorMatchDefs::BadParams
The number of images in *colorImages* does not match the number of masks in *masks*.

ccColorMatchDefs::BadImage
colorImages contains an unbound image.

ccColorMatchDefs::NotEnoughSamples
Not enough pels (< 1) in the image.

ccColorMatchDefs::BadColorSpace
colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.

ccColorMatchDefs::ShapeIsNotRegion
one of the regions in *regions* is not a region shape.

- ```
  void train(
     const cc3PlanePelBuffer& colorImage,
     const ccCompositeColorMatchTrainParams& trainParams);
  ```

Trains the *colorImage* using the *trainParams*. If the tool is already trained, it is untrained and retrained.

Parameters

colorImages The image used to train.

trainParams A **ccCompositeColorMatchTrainParams** object that specifies the sampling percentage and the size of the Gaussian smoothing kernel used to train the images.
  ```

Throws

ccColorMatchDefs::BadImage
colorImage is unbound.

ccColorMatchDefs::NotEnoughSamples
Not enough pels (< 1) in the image.

ccColorMatchDefs::BadColorSpace
colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.

untrain

```
void untrain();
```

Untrains the tool and releases any memory that was required for training. This function has no effect if the tool is already untrained.

isTrained

```
bool isTrained() const;
```

Returns True if the tool is trained; False otherwise.

run

```
void run(const cc3PlanePelBuffer &image,
         const ccCompositeColorMatchRunParams &runParams,
         ccColorMatchResult &result) const;

void run(const cc3PlanePelBuffer &image,
         const ccShapePtrh& region,
         const ccCompositeColorMatchRunParams &runParams,
         ccColorMatchResult &result) const;

void run(const cc3PlanePelBuffer &image,
         const ccPelBuffer_const<c_UInt8>& mask,
         const ccCompositeColorMatchRunParams &runParams,
         ccColorMatchResult &result) const;
```

-
```
void run(const cc3PlanePelBuffer &image,
         const ccCompositeColorMatchRunParams &runParams,
         ccColorMatchResult &result) const;
```

Runs the color match tool on *image* using the specified *runParams* and returns the results in *result*.

Parameters

image The color image.

runParams A **ccCompositeColorMatchRunParams** object the specifies the matching accuracy and whether the tool should normalize intensity

result A **ccColorMatchResult** object that contains the results of running the tool.

Throws

ccColorMatchDefs::BadImage
image is unbound.

ccColorMatchDefs::NotEnoughSamples
Not enough pels (< 1) in the image.

ccColorMatchDefs::BadColorSpace
colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.

- ```
  void run(const cc3PlanePelBuffer &image,
           const ccShapePtrh& region,
           const ccCompositeColorMatchRunParams &runParams,
           ccColorMatchResult &result) const;
  ```

Runs the color match tool on the intersection of *image* and *region* using the specified *runParams* and returns the results in *result*.

Parameters

image The color image.

region A shape that defines a region. If *region* is NULL, the whole image is used.

runParams A **ccCompositeColorMatchRunParams** object the specifies the matching accuracy and whether the tool should normalize intensity

result A **ccColorMatchResult** object that contains the results of running the tool.

Throws

ccColorMatchDefs::BadImage
image is unbound.

ccColorMatchDefs::NotEnoughSamples
Not enough pels (< 1) in the image.

ccColorMatchDefs::BadColorSpace
colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.

ccColorMatchDefs::ShapeIsNotRegion
region is not a region shape.
  ```

- ```
  void run(const cc3PlanePelBuffer &image,
           const ccPelBuffer_const<c_UInt8>& mask,
           const ccCompositeColorMatchRunParams &runParams,
           ccColorMatchResult &result) const;
  ```

  Runs the color match tool on *image* using the *mask* and the specified *runParams* and returns the results in *result*.

  Parameters

  image — The color image.

  mask — An image mask. Pels whose value is 0 are don't care pixels; pels whose values are 255 are care pixels. All other values throw an error.

  runParams — A **ccCompositeColorMatchRunParams** object the specifies the matching accuracy and whether the tool should normalize intensity

  result — A **ccColorMatchResult** object that contains the results of running the tool.

  Throws

  ccColorMatchDefs::BadImage
  image or *mask* is unbound.
  image window is not the same as *mask* window
  mask contains pel whose values are other than 0 and 255

  ccColorMatchDefs::NotEnoughSamples
  Not enough pels (< 1) in the image.

  ccColorMatchDefs::BadColorSpace
  colorImages contains an image whose color space is not **ccColorSpaceDefs::eRGB**.
  ```

ccCompositeColorMatchTool

ccCompositeColorMatchTrainParams

```
#include <ch_cvl/colmatch.h>

class ccCompositeColorMatchTrainParams
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes the parameters used to train the Composite Color Match tool.

ccCompositeColorMatchTrainParams

```
ccCompositeColorMatchTrainParams(
      double samplingPercentage,
      c_Int32 gaussianSmoothing);
```

Creates a **ccCompositeColorMatchTrainParams** object with the following default values:

Parameter	Default value
samplingPercentage	100.0
gaussianSmoothing	0

Public Member Functions

samplingPercentage

```
double samplingPercentage() const;

void samplingPercentage(double newValue);
```

- `double samplingPercentage() const;`

  Returns the sampling percentage.

- `void samplingPercentage(double newValue);`

  Sets the sampling percentage to use in composite color training.

Parameters
newValue The percentage.

Throws
ccColorMatchDefs::BadParams
newValue is greater than 100 or less than or equal to 0

gaussianSmoothing

```
c_Int32 gaussianSmoothing() const;

void gaussianSmoothing(c_Int32 newValue);
```

- `c_Int32 gaussianSmoothing() const;`

  Returns the size of the Gaussian smoothing kernel.

- `void gaussianSmoothing(c_Int32 newValue);`

  Sets the size of the Gaussian smoothing kernel.

  Parameters
  newValue The kernel size.

  Throws
  ccColorMatchDefs::BadParams
  newValue is greater than 24 or less than 0

ccConStream

```
#include <ch_cvl/constrea.h>

class ccConStream: public ccCvlOStream;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

The **ccConStream** class implements a console with which you can perform stream-based output. You can use a **ccConStream** to print text messages and other output in a scrolling text window on the host system.

You cannot use a **ccConStream** to display output on an embedded display, only on the host system.

You cannot obtain input using a **ccConStream**.

Constructors/Destructors

ccConStream

```
ccConStream(const TCHAR* consoleTitle,
  c_UInt32 consoleStyle = 0, c_UInt32 bufferSize = 20000);

ccConStream(const TCHAR* consoleTitle,
  c_UInt32 width, c_UInt32 height, c_UInt32 xPos,
  c_UInt32 yPos, c_UInt32 consoleStyle = 0,
  c_UInt32 buffersize = 20000);

~ccConStream();
```

- ```
  ccConStream(const TCHAR* consoleTitle,
    c_UInt32 consoleStyle = 0, c_UInt32 bufferSize = 20000);
  ```

  Creates a new console window at the default location and with the default size. The default console location is the upper-left corner of the screen.

  Parameters

  consoleTitle The title to display in the console window's title bar.

  consoleStyle The console style. *consoleStyle* must be formed by ORing together one or more of the following values:
  ```

ccGUI::kStyleCreateNormal
ccGUI::kStyleCreateMinimized
ccGUI::kStyleCreateMaximized
ccGUI::kStyleCreateGUI
ccGUI::kStyleCreateNow
ccGUI::kStyleCreateOnOutput
ccGUI::kStyleCloseIsHide

If *consoleStyle* does not include *ccGUI::kStyleCreateGUI*, then output sent through this **ccConStream** will be sent to **cout**, if it is available. If **cout** is not available, then a console window is created when output is sent.

If *consoleStyle* includes *ccGUI::kStyleCreateGUI*, then the console window is created immediately if *consoleStyle* includes *ccGUI::kStyleCreateNow*. If *consoleStyle* does not include *ccGUI::kStyleCreateNow*, then the console window is not created until output is sent through this **ccConStream**.

bufferSize — The number of bytes to store in the scroll back area of the console.

- ```
  ccConStream(const TCHAR* consoleTitle,
    c_UInt32 width, c_UInt32 height, c_UInt32 xPos,
    c_UInt32 yPos, c_UInt32 consoleStyle = 0,
    c_UInt32 buffersize = 20000);
  ```

  Creates a new console window at the specified location and with the specified size.

  Parameters

  consoleTitle — The title to display in the console window's title bar.

  width — The width of the console window in host display pixels. *width* includes the width of the GUI elements included with the console such as scroll bars.

  height — The height of the console window in host display pixels. *height* includes the height of the GUI elements included with the console such as scroll bars.

  xPos — The x-position of the upper-left corner of the console window in host display pixels.

  yPos — The y-position of the upper-left corner of the console window in host display pixels.

  consoleStyle — The console style. *consoleStyle* must be formed by ORing together one or more of the following values:
  ```

ccGUI::kStyleCreateNormal
ccGUI::kStyleCreateMinimized
ccGUI::kStyleCreateMaximized
ccGUI::kStyleCreateGUI
ccGUI::kStyleCreateNow
ccGUI::kStyleCreateOnOutput
ccGUI::kStyleCloseIsHide

If *consoleStyle* does not include *ccGUI::kStyleCreateGUI*, then output sent through this **ccConStream** will be sent to **cout**, if it is available. If **cout** is not available, then a console window is created when output is sent.

If *consoleStyle* includes *ccGUI::kStyleCreateGUI*, then the console window is created immediately if *consoleStyle* includes *ccGUI::kStyleCreateNow*. If *consoleStyle* does not include *ccGUI::kStyleCreateNow*, then the console window is not created until output is sent through this **ccConStream**.

bufferSize — The number of bytes to store in the scroll back area of the console.

Notes

Depending on the host operating system and environment, the console title, style, size and position arguments may or may not be used.

The **Close** command and close box in console windows created by this constructor are disabled by default. You can enable the **Close** command and close box in a console window by calling the **release()** function.

- `~ccConStream();`

Destroys and cleans up any GUI elements created to support this console. If you are transcripting this console to a **ccCvlOStream**, any buffered text is flushed to that **ccCvlOStream**.

Public Member Functions

useGUI

```
void useGUI(bool use);
bool useGUI() const;
```

- `void useGUI(bool use);`

Sets this **ccConStream** to use a GUI console window or to use **cout**. By default, a **ccConStream** sends its output to **cout**. If **cout** is not available, the **ccConStream** creates and uses a console window. You can force a **ccConStream** to create and use a console window by calling this function with a value of true.

You can also control this behavior at construction time using the *consoleStyle* argument to the constructor.

Parameters

use	If true, a console window is always created. If false, output is sent to **cout** if it is available and any existing console window is closed.

Notes

The *consoleStyle* argument to the **ccConStream** constructor determines whether the console window is created immediately or at the time that output is first sent to it.

- `bool useGUI() const;`

Returns true if this **ccConStream** is configured to always create and use a GUI console window; returns false if this **ccConStream** is configured to use **cout** if available.

release

```
void release();
```

Enables the **Close** command and close box in this console window. Until you call this function, a user cannot close a console window interactively.

You should not send any data to a **ccConStream** after calling this function.

bringToTop

```
void bringToTop();
```

Places this **ccConStream**'s console window in front of all other overlapping windows. If this **ccConStream**'s console window is minimized, it is restored, then brought to the front.

Notes

If this **ccConStream** is not using a console window, then this function has no effect.

windowRect

```
ccPelRect windowRect();
void windowRect(const ccPelRect& rect);
```

- `ccPelRect windowRect();`

  Returns a rectangle that describes the overall size and location of this **ccConStream**'s console window. The locations are in the screen pixel coordinate system.

- `void windowRect(const ccPelRect& rect);`

  Sets the overall size and location of this **ccConStream**'s console window to be the supplied rectangle. The locations are in screen pixels.

  Parameters

  rect — A **ccPelRect** describing the location of the console window in screen pixel coordinates.

  Notes

  If this **ccConStream** is not using a console window, the values are saved and used the next time a console window is created for this **ccConStream**.

minimize

```
bool minimize() const;
void minimize(bool flag);
```

- `bool minimize() const;`

  Return true if this **ccConStream**'s console window is minimized, false otherwise.

  Notes

  If this **ccConStream** is not using a console window, the value indicates whether a newly created console window will be minimized or not.

- `void minimize(bool flag);`

  Either minimizes or restores the state of this **ccConStream**'s console window.

  Parameters

  flag — If true, this **ccConStream**'s console window is minimized. If false, this **ccConStream**'s console window is restored.

  Notes

  If this **ccConStream** is not using a console window, the value determines whether a newly created console window will be minimized or not.

font

```
void font(c_UInt32 f);
c_UInt32 font(void) const;
```

- `void font(c_UInt32 f);`

  Sets the font to use to display text in this **ccConStream**'s console window. The font representation depends on the host operating system.

  Parameters

  f The font to use.

  Notes

  If this **ccConStream** is not using a console window, the value you specify is used when a new console window is created.

- `c_UInt32 font() const;`

  Returns the font used to display text in this **ccConStream**'s console window. The font representation depends on the host operating system.

transcript

```
ccCvlOStream* transcript() const;
void transcript(ccCvlOStream *stream);
```

- `ccCvlOStream* transcript() const;`

  Returns the **ccCvlOStream** to which data sent to this **ccConStream** is being transcripted.

- `void transcript(ccCvlOStream *stream);`

  Specifies that all data sent to this **ccConStream** be copied to the supplied **ccCvlOStream**.

  Parameters

  stream The **ccCvlOStream** to use. If you supply 0 for *stream*, no transcripting is performed.

clear

```
void clear(bool wait = false);
```

Clears all text from this **ccConStream**'s console window. You can specify that the text be cleared immediately or that it be cleared just before the next time output is sent to this **ccConStream**'s console window.

Parameters

wait	If true, do not clear the text until just before the next output is sent to the console window.

Notes

If this **ccConStream** is not using a console window, this function has no effect.

Functions

cfCogOut

```
ccConStream& cfCogOut();
```

Returns a **ccConStream** that refers to Cognex standard output.

cfCogDebug

```
ccConStream& cfCogDebug();
```

Returns a **ccConStream** that refers to Cognex debugging output.

ccContourTree

```
#include <ch_cvl/shaptree.h>

class ccContourTree : public ccShapeTree;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccContourTree** class is a concrete class for maintaining hierarchies of connectable (open contour) shapes, joined into a composite shape that may be open or closed. Generally, the base class methods of **ccShape** are implemented by treating the complete composite shape as a single contour, just like any primitive contour.

The children of a **ccContourTree** can be either primitive or hierarchical, but must all be open contours. Currently, the only non-primitive shapes that are open contours are themselves open **ccContourTree**s.

There is an implied ordering of primitives in a **ccContourTree** defined as the order in which the primitives, which are necessarily all leaf nodes, are encountered during a standard traversal of the tree. Two primitives are adjacent if they are adjacent in the ordered list of primitives.

The **ccContourTree** class assumes that the endpoint of one child is approximately equal to the start point of the next child, and if the **ccContourTree** is closed that the endpoint of the last child is approximately equal to the start point of the first child. Because of floating point inaccuracies, exact equality is not enforced. For example, decomposing a **ccGenRect** yields a valid **ccContourTree**, even though the rounded corner endpoints may not exactly coincide with the straight side endpoints. Such small discrepancies do not pose a problem. However, since coincidence of endpoints is not enforced, it is possible (though not advised) to create **ccContourTree**s with gaps between the endpoints of adjacent children that are large relative to the scale of the children themselves. This may produce unpredictable results, as many **ccContourTree** methods (for example, **sample()** and **isRightHanded()**) assume that the gaps are of negligible size.

Constructors/Destructors

```
explicit ccContourTree(bool closed = false);
```

Constructs a **ccContourTree** containing no children, with the given open/closed status. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

closed — If true, a closed **ccContourTree** is constructed; if false (the default), an open **ccContourTree** is constructed.

Notes

Copies and assignments are shallow, that is, they are achieved by copying pointer handles. Hence, these operations lead to sharing of children between different **ccShapeTree**s.

Operators

operator==

```
bool operator==(const ccContourTree &rhs) const;
```

Returns true if and only if this **ccContourTree** is equal to *rhs*. Two **ccContourTree**s are equal if they are equal as **ccShapeTree**s, and if they are either both open or both closed.

Public Member Functions

isClosed

```
bool isClosed() const;
void isClosed(bool closed);
```

- `bool isClosed() const;`

Gets the open/closed status of this **ccContourTree**. A **ccContourTree** is open if and only if it is not closed.

Notes

ccContourTrees are open by default.

- `void isClosed(bool closed);`

Sets the open/closed status of a **ccContourTree**.

Parameters

closed — If true, the **ccContourTree** is set to closed; if false, it is set to open.

insertChild

```
virtual void insertChild(c_Int32 idx,
  const ccShape &child);

virtual void insertChild(c_Int32 idx,
  const ccShapePtrh_const &child, bool direct = false);
```

- ```
  virtual void insertChild(c_Int32 idx,
    const ccShape &child);
  ```

  Inserts the given child into this **ccContourTree** at the given index.

  Parameters

  idx The index.

  child The child to insert.

  Throws

  ccShapesError::BadGeom
  An attempt was made to insert a child that is not an open contour.

  ccShapesError::BadIndex
  idx is less than zero or greater than **numChildren()** before the insertion.

  See **ccShapeTree::insertChild()** for more information.

- ```
  virtual void insertChild(c_Int32 idx,
    const ccShapePtrh_const &child, bool direct = false);
  ```

  Inserts the given child into this **ccContourTree** at the given index. The child is inserted by copy insertion or direct insertion, depending on the status of the *direct* flag.

  Parameters

  idx The index.

  child The child to insert.

  direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

  Throws

  ccShapesError::BadGeom
  An attempt was made to insert a child that is not an open contour.

  ccShapesError::BadIndex
  idx is less than zero or greater than **numChildren()** before the insertion.

See **ccShapeTree::insertChild()** for more information.

insertChildren

```
virtual void insertChildren(c_Int32 idx,
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Inserts the given children into this **ccContourTree** at the given index. The children are inserted by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

idx The index.

children The vector of children to insert.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom
An attempt was made to insert a child that is not an open contour.

ccShapesError::BadIndex
idx is less than zero or greater than **numChildren()** before the insertion.

See **ccShapeTree::insertChildren()** for more information.

---

addChild

```
virtual void addChild(const ccShape &child);

virtual void addChild(const ccShapePtrh_const &child,
  bool direct = false);
```

- ```
  virtual void addChild(const ccShape &child);
  ```

  Appends the given child to the end of this **ccContourTree**.

  Parameters

  child The child to add.

  Throws

  ccShapesError::BadGeom
  An attempt was made to add a child that is not an open contour.

  See **ccShapeTree:addChild()** for more information.
  ```

- ```
  virtual void addChild(const ccShapePtrh_const &child,
    bool direct = false);
  ```

  Appends the given child to the end of this **ccContourTree**. The child is added by copy insertion or direct insertion, depending on the status of the *direct* flag.

  Parameters

  child — The child to add.

  direct — If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

  Throws

  ccShapesError::BadGeom
  An attempt was made to add a child that is not an open contour.

  See **ccShapeTree::addChild()** for more information.

addChildren

```
virtual void addChildren(
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Appends the given children to the end of this **ccContourTree**. The children are added by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

children — The vector of children to add.

direct — If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom
An attempt was made to add a child that is not an open contour.

See **ccShapeTree::addChildren()** for more information.
```

replaceChild

```
virtual void replaceChild(c_Int32 idx,
  const ccShape &child);

virtual void replaceChild(c_Int32 idx,
  const ccShapePtrh_const &child, bool direct = false);
```

- ```
  virtual void replaceChild(c_Int32 idx,
    const ccShape &child);
  ```

  Replaces the child at the given index with the given child.

  Parameters

  idx The index.

  child The child replacing the indexed child.

  Throws

  ccShapesError::BadGeom
  An attempt was made to replace with a child that is not an open contour.

  ccShapesError::BadIndex
  idx is less than zero or greater than or equal to **numChildren()** before the replacement.

  See **ccShapeTree::replaceChild()** for more information.

- ```
  virtual void replaceChild(c_Int32 idx,
    const ccShapePtrh_const &child, bool direct = false);
  ```

  Replaces the indexed child with the given child. The child is replaced by copy insertion or direct insertion, depending on the status of the *direct* flag.

  Parameters

  idx The index.

  child The child replacing the indexed child.

  direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom
An attempt was made to replace with a child that is not an open contour.

ccShapesError::BadIndex
idx is less than zero or greater than or equal to **numChildren()** before the replacement.

See **ccShapeTree::replaceChild()** for more information.

replaceChildren

```
virtual void replaceChildren(
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Replaces all of the children of this **ccContourTree** with the given children. The children are replaced by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

children The vector of children replacing the current children.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom
An attempt was made to replace with a child that is not an open contour.

See **ccShapeTree::replaceChildren()** for more information.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this contour tree.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if and only if this **ccContourTree** is open overall, and has at least one primitive shape. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if and only if this **ccContourTree** is closed overall, and not empty. See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

For **ccContourTree**s, this function always returns true. All open contours are necessarily finite. See **ccShape::isFinite()** for more information.

startPoint `virtual cc2Vect startPoint() const;`

Returns the start point of the first primitive in the hierarchy if this **ccContourTree** is open.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this **ccContourTree**.

See **ccShape::startPoint()** for more information.

endPoint `virtual cc2Vect endPoint() const;`

Returns the end point of the last primitive in the hierarchy if this **ccContourTree** is open.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this **ccContourTree**.

See **ccShape::endPoint()** for more information.

startAngle `virtual ccRadian startAngle() const;`

Returns the start angle of the first primitive in the hierarchy if this **ccContourTree** is open.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this contour tree.

ccShapesError::NoTangent
hasTangent() is false for this contour tree.

See **ccShape::startAngle()** for more information.

endAngle `virtual ccRadian endAngle() const;`

Returns the end angle of the last primitive in the hierarchy if this **ccContourTree** is open.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this **ccContourTree**.

ccShapesError::NoTangent
hasTangent() is false for this contour tree.

See **ccShape::endAngle()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Reverses the individual primitives of the hierarchy, and their order within the hierarchy. See **ccShape::reverse()** for more information.

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Returns the net signed angle through which the tangent vector rotates along this **ccContourTree** from the start point to the end point.

Throws
ccShapesError::NoTangent
hasTangent() is false for this contour tree.

See **ccShape::tangentRotation()** for more information.

windingAngle

```
virtual ccRadian windingAngle(const cc2Vect &p) const;
```

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve from the start point to the end point of this **ccContourTree**.

Parameters
p The start point of the vector *p->t* whose angle is measured as the end point *t* traces the curve.

See **ccShape::windingAngle()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if the given point is within the region defined by this **ccContourTree**, if it is a closed shape.

Parameters
p The point.

Throws
ccShapesError::NotRegion
This contour tree is not a region.

See **ccShape::within()** for more information.

isRightHanded `virtual bool isRightHanded() const;`

Returns true if this **ccContourTree** is both right-handed and closed.

Throws

ccShapesError::NotRegion
This contour tree is not a region.

Notes

The return value of this function is undefined for self-intersecting **ccContourTree**s.

See **ccShape::isRightHanded()** for more information.

isReversible `virtual bool isReversible() const;`

For **ccContourTree**s, this function always returns true.

Notes

A **ccContourTree** can be reversed even when it is closed.

See **ccShape::isReversible()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this **ccContourTree** to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
 ccShape::ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this **ccContourTree**.

Parameters

params	Parameters object specifying details of how the sampling should be done.
result	Result object to which position and tangent chains are stored.

Notes

A single chain is generated for the entire **ccContourTree**.

See **ccShape::sample()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this contour tree over the given perimeter range.

Parameters

info Shape information for this contour tree.

range The perimeter range.

See **ccShape::subShape()** for more information.

ccContrastBrightnessProp

```
#include <ch_cvl/prop.h>

class ccContrastBrightnessProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a property that controls the contrast and brightness of a digitizer associated with an acquisition FIFO.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

This property does not apply to users of digital cameras on CVM6, CVM9, and CVM11. You can adjust contrast and brightness on these cameras with switches or a menu interface provided by the camera manufacturer. Consult your camera's documentation for the right procedure for your camera. Users of the Dalsa Spyder SP-xx series on CVM11 can also use **ccDigitalCameraControlProp::selectHighGain()** to adjust gain.

Contrast and brightness values control how light levels are mapped to pixel values. With 8-bit images, all light levels entering the camera must be mapped to pixel values from 0 to 255. The contrast setting determines how light levels are mapped to the 256 pixel values. Brightness determines where the center of the contrast mapping window falls relative to the range of pixel values. The mapping behavior differs depending on which analog-to-digital conversion chip (ADC) is used by the frame grabber hardware.

Fusion™ 878A Based Hardware

Cognex hardware that uses the Fusion™ 878A ADC includes the MVS-8100L. With this type of hardware, pixel value changes caused by contrast and brightness settings are applied after the A/D conversion takes place. Contrast values map to a digital multiplier, or gain, that is applied to the converted value. Brightness values map to a digital offset.

With Fusion™ 878A based hardware, *gain* is a value between 1 and 3.

Contrast	Gain
0.00	1x
0.25	1.5x
0.50	2x
0.75	2.5x
1.00	3x

The gain is calculated as follows:

*gain = 2.0 * contrast + 1.0*

With Fusion™ 878A based hardware, *offset* is a value between -128 and +127.

Brightness	Offset
0.00	-128
0.25	-64
0.50	0
0.75	+64
1.00	+127

The offset is calculated as follows:

*offset = (2 * 255 * brightness + 1)/2 - 128*

In both cases, the result is truncated to the nearest integer going towards zero. Once gain and offset values are calculated, the calculation performed to convert the digitized levels to pixel values is as follows:

pixel_value = (*digitized_level* ∗ *gain*) + *offset*

An adjusted level is first calculated from (*digitized_level* ∗ *gain*), and then an *offset* is applied to the intermediate result. At all stages of the calculation, the value must be kept within the range of 0 through 255. To keep the value within this range, a clipping algorithm is applied to both the adjusted level and the final pixel value:

adjusted_level = *digitized_level* ∗ *gain;*

if (*adjusted_level* > 255) *adjusted_level* = 255;

pixel_value = *adjusted_level* + *offset*;

if (*pixel_value* > 255) *pixel_value* = 255;

if (*pixel_value* < 0) *pixel_value* = 0;

Note The above calculation is performed entirely in hardware. It cannot be controlled in software.

TI THS8083A Based Hardware

Cognex hardware that uses the Texas Instruments THS8083A ADC includes the MVS-8501/8504, MVS-8504e, MVS-8500Le, MVS-8102, and CVM14. With these frame grabbers, contrast settings are translated into an analog gain or multiplier, and brightness settings map to a digital offset.

The THS8083A has 1.6 V as its internal full scale. This is why the minimum gain is 1.33, so that a 1.2 V input will be increased to 1.6 V. In other words, the device's designed input range is 0.4 to 1.2 V.

With THS8083A based hardware, gain is a value between 1.33 and 3.96.

Contrast	Gain
0.00	1.33x
0.25	2.0x
0.50	2.64x
0.75	3.3x
1.00	3.96x

The gain is calculated as follows:

*gain = (4/3 + (63 * contrast)/24) * (1.002)*

With THS8083A based hardware, offset is a value between -64 to +63.

Brightness	Offset
0.00	-64
0.25	-32
0.50	0
0.75	+32
1.00	+63

The offset is calculated as follows:

*offset = (brightness * 255) / 2 - 64*

TI TVP7002 Based Hardware

Cognex hardware that uses the Texas Instruments TVP7002 ADC includes the MVS-8510 grabbers (MVS-8511, 8514, 8511e, and 8514e). With these frame grabbers, contrast settings are translated into a combination of analog and digital gain, and brightness settings map to a digital offset.

The TVP7002 has 1.0 V as its internal full scale. This is why the minimum gain is 0.83, so that a 1.2 V input will be reduced to 1.0 V. The maximum gain is 2.50. In other words, the device's designed input range is 0.4 to 1.2 V.

With TVP7002 based hardware, the effective gain (the combination of analog and digital gains) is a value between 0.83 and 2.50.

Contrast	Gain
0.00	0.83x
0.25	1.25x
0.50	1.67x
0.75	2.09x
1.00	2.50x

With TVP7002 based hardware, offset is a value between -64 and +63.

Brightness	Offset
0.00	-64
0.25	-32
0.50	0
0.75	+32
1.00	+63

Constructors/Destructors

ccContrastBrightnessProp

```
ccContrastBrightnessProp();

ccContrastBrightnessProp(double contrast,
  double brightness);
```

- `ccContrastBrightnessProp();`

  Creates a new contrast and brightness property not associated with any FIFO. Brightness is set to **ccContrastBrightnessProp::defaultBrightness**. Contrast is set to **ccContrastBrightnessProp::defaultContrast**.

- `ccContrastBrightnessProp(double contrast, double brightness);`

  Creates a new contrast and brightness property not associated with any FIFO. Contrast is set to *contrast* and brightness is set to *brightness*.

  Parameters

  contrast The contrast: a value between 0.0 and 1.0.

  brightness The brightness: a value between 0.0 and 1.0.

  Throws

  ccContrastBrightnessProp::BadParams
  Value of *contrast* or *brightness* is less than 0 or greater than 1.

Enumerations

Channel

```
enum Channel;
```

This enumeration specifies the channels on a dual-tap **ccAcqFifo**.

Value	Meaning
eOdd	The input channel that produces the field that contains the first line of the image.
eEven	The input channel that produces the field that contains the second line of the image.

Public Member Functions

contrastBrightness

```
void contrastBrightness(double contrast,
  double brightness);

void contrastBrightness(double contrast,
  double brightness,
  ccContrastBrightnessProp::Channel channel);
```

- ```
  void contrastBrightness(double contrast,
    double brightness);
  ```

  Set the contrast to *contrast* and the brightness to *brightness*. If this **ccContrastBrightnessProp** is associated with a dual-tap **ccAcqFifo**, this function sets both channels to the specified brightness and contrast.

  Parameters

  contrast The contrast; a value between 0.0 and 1.0.

  brightness The brightness; a value between 0.0 and 1.0.

  Throws

  ccContrastBrightnessProp::BadParams
  Value of *contrast* or *brightness* is less than 0 or greater than 1.

- ```
  void contrastBrightness(double contrast,
    double brightness,
    ccContrastBrightnessProp::Channel channel);
  ```

  Set the contrast to *contrast* and the brightness to *brightness*.

Parameters

contrast The contrast; a value between 0.0 and 1.0.

brightness The brightness; a value between 0.0 and 1.0.

channel If this **ccContrastBrightnessProp** is associated with a dual-tap **ccAcqFifo**, *channel* specifies which tap's brightness and contrast values to set.

channel must be one of the following values:

ccContrastBrightnessProp::eEven
ccContrastBrightnessProp::eOdd

Throws

ccContrastBrightnessProp::BadParams
Value of *contrast* or *brightness* is less than 0 or greater than 1.

Notes

If you call this function on a **ccAcqFifo** associated with a single-tap camera, *ccContrastBrightnessProp::eEven* sets the brightness and contrast for the **ccAcqFifo**.

contrast

```
double contrast(ccContrastBrightessProp::Channel channel =
  ccContrastBrightnessProp::eEven);
```

Return the contrast value.

Parameters

channel If this **ccContrastBrightnessProp** is associated with a dual-tap **ccAcqFifo**, *channel* specifies which tap's contrast value is returned.

channel must be one of the following values:

ccContrastBrightnessProp::eEven
ccContrastBrightnessProp::eOdd

brightness

```
double brightness(ccContrastBrightessProp::Channel
  channel = ccContrastBrightnessProp::eEven);
```

Return the brightness value.

Parameters

channel If this **ccContrastBrightnessProp** is associated with a dual-tap **ccAcqFifo**, *channel* specifies which tap's brightness value is returned.

channel must be one of the following values:

ccContrastBrightnessProp::eEven
ccContrastBrightnessProp::eOdd

setContrast `void setContrast(double contrast);`

Set contrast only.

Parameters
contrast The new contrast.

Note **setContrast()** is not intended for use with dual-tap cameras. Use **contrastBrightness()** instead.

setBrightness `void setBrightness(double brightness);`

Sets brightness only.

Parameters
brightness The new brightness.

Note **setBrightness()** is not intended for use with dual-tap cameras. Use **contrastBrightness()** instead.

Constants

defaultContrast `static const double defaultContrast;`

The default contrast value is 0.5.

defaultBrightness
`static const double defaultBrightness;`

The default brightness value is 0.5.

ccContrastBrightnessProp

ccConvolveParams

```
#include <ch_cvl/convolve.h>

class ccConvolveParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccConvolveParams** class contains the parameters you use to control the operation of the Discrete Convolution tool (same as *Vision Tool Guide*). The Discrete Convolution tool convolves an image using a 3x3 kernel that you supply; you invoke the tool by calling the **cfConvolve()** function.

Constructors/Destructors

ccConvolveParams

```
ccConvolveParams();

explicit ccConvolveParams(ccIPair kernOriginOffset);

explicit ccConvolveParams(c_Int32 dstNorm);

ccConvolveParams(ccIPair kernOriginOffset,
  c_Int32 dstNorm);
```

- `ccConvolveParams();`

  Constructs a **ccConvolveParams** object with default parameter values (automatic kernel prescaling and normalization and a kernel origin in the center of the kernel).

- `explicit ccConvolveParams(ccIPair kernOriginOffset)`

  Constructs a **ccConvolveParams** that specifies automatic kernel prescaling and normalization and the supplied kernel origin.

  You specify the kernel origin by supplying a **ccIPair** interpreted as shown in the following figure:

-1, -1	0, -1	1, -1
-1, 0	0, 0	1, 0
-1, 1	0, 1	1, 1

  Parameters

  kernOriginOffset The kernel origin.

- `explicit ccConvolveParams(c_Int32 dstNorm);`

  Constructs a **ccConvolveParams** that specifies a manual normalization factor and a kernel origin in the center of the kernel.

  Parameters

  dstNorm The normalization factor. The result of applying the kernel is right-shifted by the specified number of bits.

- `ccConvolveParams(ccIPair kernOriginOffset, c_Int32 dstNorm);`

  Constructs a **ccConvolveParams** that specifies a manual normalization factor and the supplied kernel origin.

  You specify the kernel origin by supplying a **ccIPair** interpreted as shown in the following figure:

-1, -1	0, -1	1, -1
-1, 0	0, 0	1, 0
-1, 1	0, 1	1, 1

Parameters

kernOriginOffset The kernel origin.

dstNorm The normalization factor. The result of applying the kernel is right-shifted by the specified number of bits.

Public Member Functions

originOffset

```
ccIPair originOffset() const;
void originOffset(ccIPair kernOriginOffset);
```

- `ccIPair originOffset() const;`

  Returns the kernel origin specified by this **ccConvolveParams**.

- `void originOffset(ccIPair kernOriginOffset);`

  Sets the kernel origin specified by this **ccConvolveParams**.

  Parameters

  kernOriginOffset A **ccIPair** that specifies the kernel origin. A value of 0,0 indicates the center of the kernel.

isNormalized

```
bool isNormalized() const;
```

Returns true if this **ccConvolveParams** is configured to perform automatic kernel prescaling and normalization, false if it is configured to use a manual normalization factor.

norm

```
c_Int32 norm() const;
```

```
void norm(c_Int32 dstNorm);
```

- `c_Int32 norm() const;`

  Returns the manual normalization factor specified for this **ccConvolveParams**. If this **ccConvolveParams** is configured to perform automatic kernel prescaling and normalization, the value returned by this function is undefined.

- `void norm(c_Int32 dstNorm);`

  Sets the manual normalization factor for this **ccConvolveParams**. The result of applying the kernel is right-shifted by the specified number of bits to compute the output pixel value.

  Parameters

  dstNorm The normalization factor.

  Notes

  This function has no effect if this **ccConvolveParams** is configured to perform automatic kernel prescaling and normalization.

ccCoordAxes

```
#include <ch_cvl/simpshap.h>

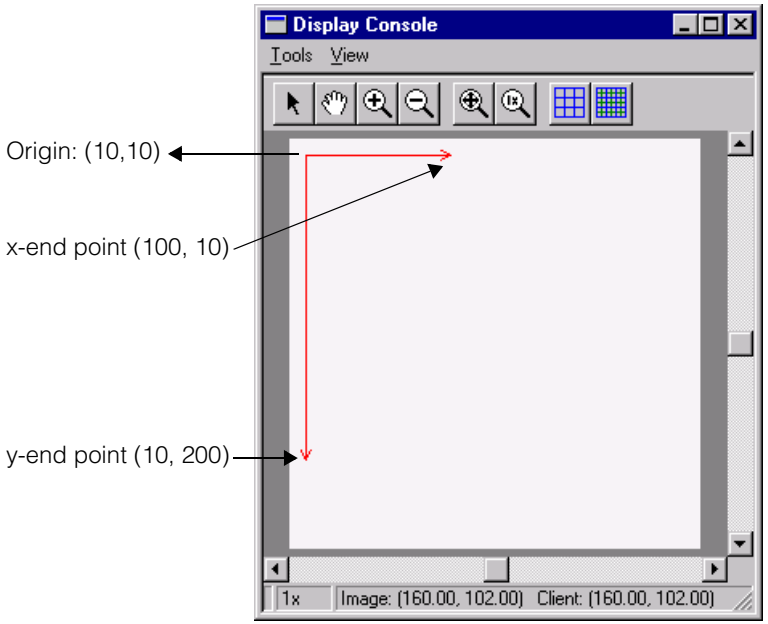
class ccCoordAxes;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

Use this class to draw coordinate axes by specifying an origin point and the end points for each axis. The following figure illustrates a **ccCoordAxes** shape.

```
ccCoordAxes(cc2Vect(10,10), cc2Vect(100,10), cc2Vect(10,200))
```



Note **ccCoordAxes** is one of the few shapes that does not inherit from **ccShape**.

Constructors/Destructors

ccCoordAxes

```
ccCoordAxes();

ccCoordAxes(const cc2Vect& or, const cc2Vect& x,
  const cc2Vect& y);
```

- `ccCoordAxes();`

  The default constructor creates a degenerate coordinate axis object whose origin, x-axis end point and y-axis end point are all (0,0).

- `ccCoordAxes(const cc2Vect& or, const cc2Vect& x, const cc2Vect& y);`

  Creates a coordinate axis object with the specified origin and end points.

  Parameters

or	The origin.
x	The x-axis end point.
y	The y-axis end point.

Operators

operator==

```
bool operator==(const ccCoordAxes& other) const;
```

Returns true if this coordinate axis object is equal to another one.

Parameters

other	The other coordinate axis object.

operator!=

```
bool operator!=(const ccCoordAxes& other) const;
```

Returns true if this coordinate axis object is not equal to another one.

Parameters

other	The other coordinate axis object.

Public Member Functions

origin

```
const ccPoint& origin() const;
```

Returns the origin of this coordinate axis object.

xEnd

```
const ccPoint& xEnd () const;
```

Returns the x-axis end point of this coordinate axis object.

yEnd

```
const ccPoint& yEnd () const;
```

Returns the y-axis end point of this coordinate axis object.

xAxis

```
ccLineSeg xAxis() const;
```

Returns the line segment from the origin to the x-axis end point.

yAxis

```
ccLineSeg yAxis() const;
```

Returns the line segment from the origin to the y-axis end point.

map

```
ccCoordAxes map(const cc2Xform& c) const;
```

Returns a coordinate axis object that is the result of mapping this object with the transformation object *c*.

Parameters

c	The transformation object.

distanceToPoint

```
double distanceToPoint(const cc2Vect& v) const;
```

Returns the nearest point on these coordinate axes to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

v	The point.

boundingBox

```
ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses these coordinate axes. See **ccShape::boundingBox()** for more information.

degen

```
bool degen() const;
```

Returns true if the distance from the origin to the x-axis end point or the y-axis end point is zero.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

ccCriticalSection

```
#include <ch_cvl/threads.h>

class ccCriticalSection;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A critical section is an object that only one thread of execution can own at a time. A thread may claim the critical section as many times as it wishes, but it must release the critical section once for every time it claims the critical section. A thread that tries to claim a critical section that is already claimed by some other thread will block until the critical section becomes available. A thread usually claims a critical section by using a **ccCriticalSectionLock** object.

Critical sections are generally faster to claim and release than mutexes. However, there is no timeout available on lock, named critical sections are not available, and a critical section does not throw *BrokenLock*.

Constructors/Destructors

ccCriticalSection

```
ccCriticalSection();
```

Constructs a new **ccCriticalSection**.

~ccCriticalSection

```
~ccCriticalSection();
```

Destroys a **ccCriticalSection**.

Public Member Functions

lock

```
void lock();
```

Locks the **ccCriticalSection**. Any other thread that attempts to lock this critical section will block until it is released.

unlock

```
void unlock();
```

Unlocks this **ccCriticalSection**.

Notes

ccCriticalSection::unlock() must be called as many times as **ccCriticalSection::lock()**.

ccCriticalSectionLock

```
#include <ch_cvl/threads.h>

class ccCriticalSection;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A critical section lock sets a lock on a critical section. The destructor automatically unlocks, so unlocking is assured.

Constructors/Destructors

ccCriticalSectionLock

```
ccCriticalSectionLock(ccCriticalSection& cs,
  bool locked = true);
```

Constructs a new **ccCriticalSectionLock**.

Parameters

cs	The critical section to be locked.
locked	If *true* (the default), the critical section is claimed, possibly blocking access to it by others indefinitely. If *false*, the critical section is not immediately claimed.

~ccCriticalSectionLock

```
~ccCriticalSectionLock();
```

Destroys a **ccCriticalSectionLock** object, undoing all outstanding lock operations done by this object.

Notes

At the time of the critical section lock's destruction, if there have been fewer unlock invocations than lock invocations, the destructor invokes **ccCriticalSectionLock::unlock()** enough times to make up the difference.

Public Member Functions

lock `void lock();`

Locks the critical section. Any other thread that attempts to lock the same critical section will block until the critical section is released.

unlock `void unlock();`

Unlocks the critical section.

Notes

ccCriticalSectionLock::unlock() must be called as many times as **ccCriticalSectionLock::lock()**.

ccCross

```
#include <ch_cvl/simpshap.h>

class ccCross;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a rotated cross with a fixed arm length. The arm length is constant when the cross is displayed in different coordinate systems or zoomed in a display console. **ccCross** objects are primarily used in drawing vision tool results. They are not generally used as geometric objects for describing shapes.

Note **ccCross** is one of the few shapes that does not inherit from **ccShape**.

Constructors/Destructors

ccCross

```
ccCross();

ccCross(const ccPoint& or, const ccRadian& angle,
  int armLength = 7);
```

- `ccCross();`

  Default constructor. Creates a degenerate cross with origin at (0,0), an angle of 0°, and an arm length of 0.

- `ccCross(const ccPoint& or, ccRadian& angle, int armLength = 7);`

  Constructs a cross with the specified origin, angle, and arm length.

  Parameters

or	The origin specified as a **ccPoint** object.
angle	The angle of rotation specified in radians as a **ccRadian** object.
armLength	The arm length specified in pixels (default = 7 pixels).

Operators

operator==

```
bool operator==(const ccCross& that) const;
```

Returns true if this cross is equal to *that*, false otherwise.

Parameters

that	The other cross.

operator!=

```
bool operator!=(const ccCross& that) const;
```

Returns true if this cross is not equal to *that*, false otherwise.

Parameters

that	The other cross.

Public Member Functions

origin

```
const ccPoint& origin() const;
void origin(const ccPoint& or);
```

- `const ccPoint& origin() const;`

  Returns the origin of the cross as a **ccPoint**.

- `void origin(const ccPoint& or);`

  Sets the origin of the cross.

  Parameters

or	Origin of the cross specified as a **ccPoint** containing the x and y coordinates.

angle

```
void angle(const ccRadian& angle);
ccRadian& angle() const;
```

- `void angle(const ccRadian& angle);`

  Sets the angle of the cross.

  Parameters

angle	Angle of rotation of the cross, specified in radians as a **ccRadian** object.

- `ccRadian& angle() const;`

  Returns the angle of rotation of the cross in radians as a **ccRadian** object.

armLength

```
void armLength(int armLength);
int armLength() const;
```

- `void armLength(int armLength);`

  Sets the arm length of the cross.

  Parameters

  armLength Arm length of the cross specified in pixels.

- `int armLength() const;`

  Returns the arm length of the cross in pixels as a non-const integer.

map

```
ccCross map(const cc2Xform& c) const;
```

Returns the cross mapped by the specified transform. The arm length is not mapped.

Parameters

c The transformation object.

ccCross

ccCubicSpline

```
#include <ch_cvl/spline.h>

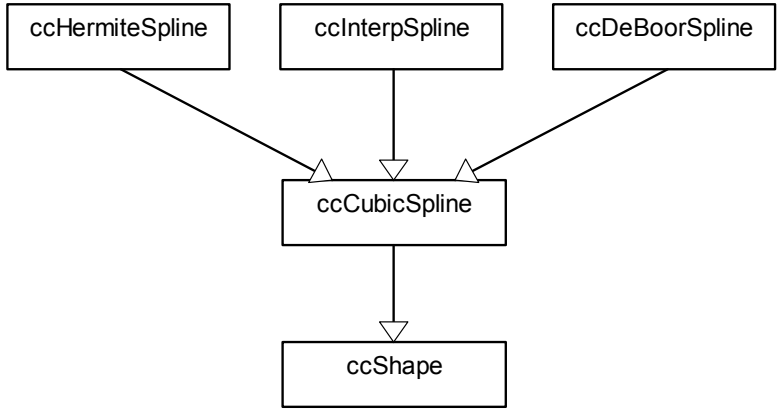
class ccCubicSpline : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccCubicSpline** class is an abstract base class for classes that implement various types of 2D cubic splines.

There are many flavors of splines (including Hermite splines, interpolation splines, and De Boor splines), but almost all of them can be viewed as a sequence of component Bezier curves joined end to end. The **ccCubicSpline** class supports examination of the component cubic Bezier curves; querying and manipulation of the parameterization via intervals; interrogation of the spline curve, such as finding points and tangents along it; and some basic geometric queries, such as finding the nearest point and intersection.



The above figure shows the **ccCubicSpline** class inheritance hierarchy.

Control Points

Like Bezier curves, splines are defined by control points. Spline control points are not the same points as the Bezier control points of the component Bezier curves. Rather, the spline control points and other spline control data determine the Bezier control points. Although n individual Bezier curves require 4n Bezier control points, a spline comprising n Bezier curves typically has fewer spline control points, often on the order of one spline control point per Bezier curve. Differentiability constraints imposed between adjacent Bezier curves supply the extra information needed to generate the Bezier control points. This reduction in degrees of freedom makes the spline easier to manipulate, and also forces it to be the sort of well-behaved, smooth curve that is useful in modeling and design.

Parameterizations

An important aspect of a cubic spline is its parameterization. Imagine a point [x(*t*), y(*t*)] moving along an entire spline. The parameterization determines what fraction of the total time the point spends on each component Bezier curve. One way of specifying a parameterization is through a knot vector. This is a non-decreasing sequence of scalar values, starting at zero. As the parameterized point moves along spline, it travels along the *i*'th Bezier curve while *t* is between the *i*'th and (*i*+1)th knot values. The knots are where the parameterized point moves from one Bezier curve to another.

For example, consider a spline comprising four Bezier curves with the knot vector (0, 2, 6, 7, 8). Then the parameterized point [x(*t*), y(*t*)] traverses the curve as follows:

1. It traverses the first curve for 0 <= *t* <= 2.
2. It traverses the second curve for 2 <= *t* <= 6.
3. It traverses the third curve for 6 <= *t* <= 7.
4. Finally, it traverses the fourth curve for 7 <= *t* <= 8.

For the spline classes in CVL, it is more convenient to work with intervals rather than knots. An interval vector is a sequence of non-negative scalar values, interval *i* corresponding to the difference between the *i*'th and (*i+1*)th knot. Thus, there is one interval for each Bezier curve. For this example, the interval vector is (2, 4, 1, 1).

Changing the parameterization actually changes the shape of the curve in order to maintain differentiability constraints at the junctions between Bezier curves. Certain parameterizations may produce better shaped curves than others. If the designer would rather not think about parameterizations, there are several standard strategies for generating them automatically:

Parameterization	Intervals Description
Uniform	All equal to 1.0
Chord length	Proportional to the Euclidean distance between relevant control points
Centripetal	Proportional to square root of Euclidean distance between relevant control points
Fixed	Not modified

The fixed parameterization is described in the section *Interval Mode* on page 1111).

The chord length and centripetal parameterizations, which are based on the geometry of the control points, typically give better-shaped splines than uniform parameterization. Centripetal parameterization is often preferred, as it tends to produce less *loopy* splines than chord length parameterization. One distinct advantage of uniform parameterization is that it alone is invariant under affine transformation of the spline control points. This has certain advantages when mapping splines (see *Reparameterizing Splines* on page 1112).

Interval Mode

The interval mode specifies the parameterization strategy used for a cubic spline. Changing the interval mode does not in itself affect any of the intervals. However, the interval mode does determine how the intervals are adjusted when you manipulate control points. When you add, remove, or move control points, the intervals that depend on those control points are automatically recomputed according to the current interval mode; the other intervals remain unchanged.

Individual intervals may also be adjusted manually. Hence, there is a distinction between the current interval mode, which is always one of the valid strategies, and the actual intervals of the spline, which may correspond to the current interval mode, or some combination of interval modes, or some completely arbitrary parameterization. The spline's parameterization is considered *conforming* if and only if all intervals correspond to the current interval mode. This is the case by default. A parameterization can only be made non-conforming by doing one of the following:

- Changing the interval mode of a non-empty spline
- Manually adjusting intervals

- Generating a new spline by mapping (see *Mapping Splines* below)

The fixed parameterization is similar to the uniform one, except that intervals are not adjusted when control points are moved. When control points are added or removed, or when the entire parameterization is recomputed using **reparameterize()**, the relevant intervals are computed as if the parameterization were uniform.

Mapping Splines

The **map()** method operates by transforming all spline control points by a supplied transform. It does *not* adjust the parameterization. The advantage of this behavior is that the locus of points of the mapped spline is exactly the locus of points of the original spline mapped by the supplied transform. More precisely, for an arbitrary spline *s*, transform *X*, and parameter *t*:

X.mapPoint(s.point(t)) == s.map(X).point(t)

within floating point tolerances. This is the behavior of the **map()** methods of all other shapes as well. The unavoidable disadvantage is that a mapped spline may not have a conforming parameterization, even if the original one does. For example, mapping by a non-rigid affine transform changes the distances between control points. If the transform has shear or aspect, even the relative distances change. In these cases, even if the original spline has, for example, a conforming chord-length parameterization, the mapped spline will not. If the control points of the mapped spline are subsequently manipulated, affected intervals will be automatically recomputed according to the current interval mode. To avoid surprising shape changes, the mapped spline may need to be reparameterized (see *Reparameterizing Splines* on page 1112).

Reparameterizing Splines

All spline classes provide a **reparameterize()** method to recompute all intervals according to the current interval mode and control point positions. This reestablishes a conforming parameterization, although it may alter the shape of the spline curve. It is sometimes desirable to invoke **reparameterize()** on a newly mapped spline to restore conformance. There are two cases in which invoking **reparameterize()** on a mapped spline is guaranteed to leave the intervals (and therefore the shape of the curve) unchanged:

- The first is when the original spline had a conforming uniform parameterization.
- The second is when the original spline had a conforming parameterization of any kind and the mapping transform was rigid.

Reparameterizing a spline after mappings that only involve translation, rotation, and uniform scaling also leaves the shape of the spline curve unchanged in many cases. Most splines are invariant with respect to a uniform scaling of the parameterization: multiplying all intervals by a constant factor does not change the curve. Exceptions to this rule are splines that have control knobs involving tangent vectors, such as Hermite

splines, and C^2 interpolation splines with clamped end conditions (see below). Since these tangent vectors are derivatives with respect to the parameter, scaling the parameterization changes the shape of the curve.

Global and Local Parameterization

Under global parameterization, points along a spline are given by two coordinate functions, x(*t*) and y(*t*), over the parameter range 0 through *M*. The values *t* = 0 and *t* = *M* correspond to the start and end points of the spline, respectively. *M* is the sum of the spline's intervals, and is available through the **maxParam()** method. All spline methods that take or return a single parameter value use global parameterization unless otherwise noted. Also, parametric tangent vectors are determined with respect to global parameterization. It is sometimes convenient to use an alternative scheme for referencing points on the spline. This scheme uses an integer index and a local parameter in the range 0 through 1. The index specifies a particular component Bezier curve, and the local parameter specifies a point within that Bezier curve. The **ccCubicSpline** class provides methods for converting between local and global parameter values.

B-Splines

Cubic B-splines are a particular type of cubic spline for which the functions x(*t*) and y(*t*) are generally twice differentiable at the junction between two cubic Bezier curves (that is, at the knot values). Along the Bezier curves, between the knots, x(*t*) and y(*t*) are infinitely differentiable, as always. Thus, the overall curve is C^2.

The B-spline coordinate functions x(*t*) and y(*t*) are always continuous, but they can exhibit less than C^2 differentiability when there are intervals of length zero, which occur when knot values are repeated. If an interval is zero (that is, when two consecutive knot values are equal), x(*t*) and y(*t*) are only once differentiable at the knot. This is the case, for example, where a straight segment tangentially meets a circular arc. If two or more consecutive intervals are zero (that is, when three or more consecutive knots are equal), x(*t*) and y(*t*) are not differentiable at the knot. This is the case, for example, where two straight segments meet at a corner.

B-splines may be open or closed. For closed B-splines, there is a C^2 connection between the first and last Bezier curves unless knots are repeated.

CVL supports two types of cubic B-splines: de Boor splines (implemented with the **ccDeBoorSpline** class) and C^2 interpolation splines (implemented with the **ccInterpSpline** class). The two types differ in how the spline control points, along with additional shaping parameters, are used to generate the Bezier control points.

Note For in-depth information on the theory and use of cubic Bezier curves and splines, see any textbook on the subject, such as *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, 1990, ISBN 0-12-249051-7.

Note All methods defined for this class leave the cubic spline object unchanged when they throw an error.

Enumerations

IntervalMode `enum IntervalMode;`

Interval mode is one of the parameterization strategies used with cubic splines. See *Interval Mode* on page 1111 for more information.

Parameters

eUniform	Parameterization intervals are all equal.
eChordLength	Parameterization intervals are proportional to the Euclidean distance between relevant control points.
eCentripetal	Parameterization intervals are proportional to the square root of the Euclidean distance between relevant control points.
eFixed	Parameterization intervals are not modified when control points are moved.

Operators

operator== `bool operator==(const ccCubicSpline &rhs) const;`

Returns true if and only if this **ccCubicSpline** is equal to *rhs*. Two splines are equal if all of their defining parameters are identical (no tolerance is used).

Parameters

rhs	The other **ccCubicSpline**.

Notes

Splines that describe identical curves are not necessarily equal.

operator!= `bool operator!=(const ccCubicSpline &rhs) const;`

Returns the opposite truth value of **operator==()**.

Constructors/Destructors

ccCubicSpline

```
explicit
ccCubicSpline(bool isClosed = false,
  IntervalMode intervalMode = eUniform);
```

Constructs an empty cubic spline, that is, one with no Bezier curves. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

isClosed The open/closed state of this cubic spline (default = open).

intervalMode The interval mode. Must be one of the following:
ccCubicSpline::eUniform (default)
ccCubicSpline:eChordLength
ccCubicSpline::eCentripetal
ccCubicSpline::eFixed

Notes

As a **ccCubicSpline** is an abstract class, only derived class constructors can call this constructor.

Public Member Functions

intervalMode

```
IntervalMode intervalMode() const;

virtual void intervalMode(IntervalMode intervalMode);
```

- `IntervalMode intervalMode() const;`

  Gets the interval mode of this spline.

- `virtual void intervalMode(IntervalMode intervalMode);`

  Sets the interval mode of this spline.

  Parameters

  intervalMode The interval mode. Must be one of the following:
  ccCubicSpline::eUniform (default)
  ccCubicSpline:eChordLength
  ccCubicSpline::eCentripetal
  ccCubicSpline::eFixed

Throws

ccShapesError::BadParams

The specified interval mode is not valid for this **ccCubicSpline**. Derived spline classes are not required to support all possible interval modes. Those that do not should override the default method, which never throws, and should indicate which interval modes are valid.

---

interval

```
double interval(c_Int32 idx) const;

void interval(c_Int32 idx, double val);
```

- `double interval(c_Int32 idx) const;`

Gets the interval with the given index.

Parameters

idx The index.

Throws

ccShapesError::BadIndex

idx is less than zero, or greater than or equal to **numBezierCurves()**.

- `void interval(c_Int32 idx, double val);`

Sets the interval with given index.

Parameters

idx The index.

val The interval.

Throws

ccShapesError::BadIndex

idx is less than zero or greater than or equal to **numBezierCurves()**.

ccShapesError::BadParams

The new interval is negative.

intervals

```
const cmStd vector<double> &intervals() const;
void intervals(const cmStd vector<double> &vals);
```

- `const cmStd vector<double> &intervals() const;`

  Gets the interval vector.

- `void intervals(const cmStd vector<double> &vals);`

  Sets the interval vector.

  Parameters

  val The interval vector.

  Throws

  ccShapesError::BadParams
  The length of the new interval vector is not equal to **numBezierCurves()**.

  ccShapesError::BadParams
  Any of the new intervals are negative.

maxParam

```
double maxParam() const;
```

Returns the global parameter value of the end point of the spline. This is the sum of all the intervals.

Notes

Points along a spline are parameterized over the range 0 through **maxParam()**.

globalParam

```
double globalParam(c_Int32 idx, double localParam) const;
```

Given the index of a Bezier curve and a local parameter in the range 0 through 1 along that curve, this function returns the corresponding global spline parameter in the range 0 through **maxParam()**.

Parameters

idx The index of one of the component Bezier curves.

localParam The local parameter. Must be in the range 0 through 1.

Notes

localParam is internally clipped if it falls outside the range 0 through 1.

Throws

ccShapesError::BadIndex
idx is less than zero or greater than or equal to **numBezierCurves()**.

ccShapesError::EmptyShape
This spline is empty.

localParam

```
c_Int32 localParam(double globalParam, double &localParam)
  const;
```

Given a global parameter along the spline in the range 0 through **maxParam()**, this function returns the corresponding Bezier curve index and local parameter in the range 0 through 1.

Parameters

globalParam	The global parameter along this spline.
localParam	The returned local parameter.

Notes

globalParam is internally clipped if it falls outside the range 0 through **maxParam()**.

Throws

ccShapesError::EmptyShape
This spline is empty.

reparameterize

```
virtual void reparameterize() = 0;
```

Recomputes all intervals based on the current control points and interval mode.

Notes

Invoking this method reestablishes a conforming parameterization, but may alter the shape of the spline curve. See *Reparameterizing Splines* on page 1112.

numControlPoints

```
c_Int32 numControlPoints() const;
```

Returns the number of control points of this spline.

isClosed

```
bool isClosed() const;

virtual void isClosed(bool) = 0;
```

- `bool isClosed() const;`

  Gets the open/closed state of this spline (true = closed, false = open).

- `virtual void isClosed(bool) = 0;`

  Sets the open/closed state of this spline.

  Parameters

  bool True sets the spline to closed, false sets it to open.

  Notes

  This setter automatically invokes **reparameterize()** to recompute the entire parameterization if and only if the open/closed state of the spline is changed.

controlPoint

```
cc2Vect controlPoint(c_Int32 idx) const;

virtual void controlPoint(c_Int32 idx,
  const cc2Vect &ctrlPt) = 0;
```

- `cc2Vect controlPoint(c_Int32 idx) const;`

  Gets the control point with the given index.

  Parameters

  idx The index.

  Throws

  ccShapesError::BadIndex

  idx is less than 0, or greater than or equal to **numControlPoints()**.

- `virtual void controlPoint(c_Int32 idx, const cc2Vect &ctrlPt) = 0;`

  Sets the control point with given index.

  Parameters

  idx The index.

  ctrlPt The control point.

Notes

Affected intervals are recomputed according to the current interval mode.

Throws

ccShapesError::BadIndex

idx is less than 0, or greater than or equal to **numControlPoints()**.

---

controlPoints

```
const cmStd vector<cc2Vect> &controlPoints() const;

virtual void controlPoints(
  const cmStd vector<cc2Vect> &ctrlPts) = 0;
```

- ```
  const cmStd vector<cc2Vect> &controlPoints() const;
  ```

  Gets the entire vector of control points.

- ```
  virtual void controlPoints(
    const cmStd vector<cc2Vect> &ctrlPts) = 0;
  ```

  Sets the entire vector of control points.

  Parameters

  ctrlPts The vector of control points.

  Notes

  This setter automatically invokes **reparameterize()** to recompute the entire parameterization.

insertControlPoint

```
virtual void insertControlPoint(c_Int32 idx,
  const cc2Vect &point) = 0;
```

Inserts a control point into the sequence of control points. The new point is inserted before the current control point with given index. The new point has index *idx* after the insertion, and the indices of all control points that were previously indexed *idx* or higher are incremented. Affected intervals are recomputed according to the current interval mode.

Parameters

idx The index.

Index Value	Effect
0	Inserts the control point at the beginning of the sequence
numControlPoints()	Inserts the control point at the end of the sequence

Throws

ccShapesError::BadIndex

idx is less than 0, or greater than **numControlPoints()** before the insertion.

addControlPoint `void addControlPoint(const cc2Vect &point);`

Convenience function. Appends a control point at the end of the sequence of control points. Equivalent to **insertControlPoint(numControlPoints(), point)**.

Parameters

point The control point to insert.

removeControlPoint

```
virtual void removeControlPoint(c_Int32 idx) = 0;
```

Removes the control point with given index.

Parameters

idx The index.

Notes

Affected intervals are recomputed according to the current interval mode.

Throws

ccShapesError::BadIndex

idx is less than 0, or greater than or equal to **numControlPoints()** before the removal.

numBezierCurves

```
c_Int32 numBezierCurves() const;
```

Returns the number of component Bezier curves of this spline.

bezierCurve `const ccBezierCurve &bezierCurve(c_Int32 idx) const;`

Returns the component Bezier curve with given index.

Parameters

idx The index.

Throws

ccShapesError::BadIndex

idx is less than 0 or greater than or equal to **numBezierCurves()**.

point `cc2Vect point(double t) const;`

Returns the point at global parameter value *t* along the spline. *t* is internally clipped if it falls outside the range 0 through **maxParam()**. The value returned is an exact value, not an approximation.

Parameters

t The global parameter value along this spline.

Notes

Splines are not parameterized by arc length, so sampling at equally spaced values of *t* does not correspond to sampling uniformly along the curve.

Throws

ccShapesError::EmptyShape

This spline is empty.

tangent `cc2Vect tangent(double t) const;`

Parameters

t The global parameter value along this spline.

Returns the tangent vector at global parameter value *t* along this spline. *t* is internally clipped if it falls outside the range 0 through **maxParam()**. The value returned is an exact value, not an approximation.

Notes

The returned tangent vector is a derivative with respect to the global spline parameter, not with respect to the local (unit interval) parameter on a component Bezier curve.

Throws

ccShapesError::NoTangent

A tangent vector does not exist at the given parameter value.

ccShapesError::EmptyShape

This spline is empty.

pointAndTangent

```
void pointAndTangent(double t, cc2Vect &point,
  cc2Vect &tangent) const;
```

Returns the point and tangent vector at global parameter value *t* along this spline. *t* is internally clipped if it falls outside the range 0 through **maxParam()**. The value returned is an exact value, not an approximation.

Parameters

t	The global parameter value along this spline.
point	Return parameter for the point computed by this function.
tangent	Return parameter for the tangent vector computed by this function.

Notes

The returned tangent vector is a derivative with respect to the global spline parameter, not with respect to the local (unit interval) parameter on a component Bezier curve.

Throws

ccShapesError::NoTangent
No tangent vector exists at the given parameter value.

ccShapesError::EmptyShape
This spline is empty.

nearestPoint

```
cc2Vect nearestPoint(const cc2Vect& p, double &t) const;

virtual cc2Vect nearestPoint(const cc2Vect& p) const;
```

- `cc2Vect nearestPoint(const cc2Vect& p, double &t) const;`

Returns the nearest point on this cubic spline to the given point as well as the nearest point's global parameter value in the range 0 through **maxParam()**.

Parameters

p	The point.
t	Return parameter for the nearest point's global parameter value.

Throws

ccShapesError::EmptyShape
This spline is empty.

See **ccShape::nearestPoint()** for more information.

- `virtual cc2Vect nearestPoint(const cc2Vect& p) const;`

  Returns the nearest point on this cubic spline to the given point. If the nearest point is not unique, one of the nearest points is returned.

  Parameters

  p The point.

  Throws

  ccShapesError::EmptyShape
  This spline is empty.

  See **ccShape::nearestPoint()** for more information.

intersections

```
cmStd vector<cc2Vect> intersections(const ccLineSeg &seg)
 const;
```

Returns the global parameter values at which this spline curve intersects the given line segment.

Parameters

seg The line segment.

Notes

If a portion of the spline curve lies exactly on the line segment, then there are infinite intersections. In this case, only a single intersection from this infinite set is returned.

This function returns an empty vector if this cubic spline is empty.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if both **isClosed()** and **isEmpty()** are false for this cubic spline. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if **isClosed()** is true and **isEmpty()** is false for this cubic spline. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For cubic splines, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if this cubic spline does not have enough points to define at least one Bezier curve. The minimum number of points required to define at least one Bezier curve depends on the type of spline and whether it is open or closed. For example, an open **ccDeBoorSpline** is empty if it has fewer than four control points; a closed **ccDeBoorSpline** is empty if it has fewer than three control points.

See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

Returns true if this cubic spline has at least one Bezier curve for which **hasTangent()** is true. See **ccShape::hasTangent()** for more information.

isReversible `virtual bool isReversible() const;`

For cubic splines, this function always returns true. See **ccShape::reverse()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For cubic splines, this function always returns false. See **ccShape::isDecomposed()** for more information.

startPoint `virtual cc2Vect startPoint() const;`

Returns the starting point of this cubic spline.

Throws

ccShapesError::NotOpenContour
This cubic spline is not an open contour.

See **ccShape::startPoint()** for more information.

endPoint `virtual cc2Vect endPoint() const;`

Returns the ending point of this cubic spline.

Throws

ccShapesError::NotOpenContour
This cubic spline is not an open contour.

See **ccShape::endPoint()** for more information.

startAngle `virtual ccRadian startAngle() const;`

Returns the starting angle of this cubic spline.

Throws

ccShapesError::NotOpenContour
This cubic spline is not an open contour.

ccShapesError::NoTangent
hasTangent() is false for this cubic spline.

See **ccShape::startAngle()** for more information.

endAngle `virtual ccRadian endAngle() const;`

Returns the ending angle of this cubic spline.

Throws

ccShapesError::NotOpenContour
This cubic spline is not an open contour.

ccShapesError::NoTangent
hasTangent() is false for this cubic spline.

See **ccShape::endAngle()** for more information.

tangentRotation `virtual ccRadian tangentRotation() const;`

Returns the net signed angle through which the tangent vector rotates along this cubic spline from the start point to the end point.

Throws

ccShapesError::NotOpenContour
This cubic spline is not an open contour.

ccShapesError::NoTangent
hasTangent() is false for this cubic spline.

See **ccShape::tangentRotation()** for more information.

windingAngle `virtual ccRadian windingAngle(const cc2Vect &p) const;`

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve.

Parameters

p The start point of the vector *p->t* whose angle is measured as the end point *t* traces the curve.

Throws

ccShapesError::NotOpenContour
This cubic spline is not an open contour.

See **ccShape::windingAngle()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

Returns true if this cubic spline is both a region and right handed.

Notes

The return value is undefined for self-intersecting cubic splines.

Throws

ccShapesError::NotRegion
This cubic spline is not a region.

See **ccShape::isRightHanded()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if the given point is within the region defined by this cubic spline.

Parameters

p The point.

Throws

ccShapesError::NotRegion
This cubic spline is not a region.

See **ccShape::within()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this cubic spline.

Notes

This function returns a tight rectangle based on the curve itself, not the control points.

Throws

ccShapesError::EmptyShape
This cubic spline is empty.

See **ccShape::boundingBox()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this **ccCubicSpline**. Returned sample points are generally not equally spaced along the curve.

Parameters

params Parameters object specifying details of how the sampling should be done.

result Result object to which position and tangent chains are stored.

Notes

If **params.computeTangents()** is true, this function ignores cubic splines for which **hasTangent()** is false.

Throws

ccShapesError::SampleOverflow

Supplied spacing and tolerance bounds require more than *maxPoint* samples to be generated. See **ccSampleParams** for details.

See **ccShape::sample()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccBezierCurve**s. See **ccShape::decompose()** for more information.

ccCustomProp

```
#include <ch_cvl/prop.h>

class ccCustomProp : virtual public ccPersistent ;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class encapsulates a collection of property values associated with an acquisition FIFO. You use this class to specify FIFO properties for devices such as GigE cameras.

Like other properties, the properties in this class are written to the hardware before an image is acquired when either of the following conditions are true:

- The properties have been changed since the last image was acquired.
- Another acquisition FIFO has written its properties to the hardware since the last image was acquired through this FIFO.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccCustomProp

```
ccCustomProp();
```

Constructs a **ccCustomProp** with no property values.

Public Member Functions

customValues

```
void customValues(ccCustomPropertyBag values);

ccCustomPropertyBag customValues() const;
```

- ```
  void customValues(ccCustomPropertyBag values);
  ```

  Sets the properties and values for this FIFO.
  ```

Parameters

values	A **ccCustomPropertyBag** containing the properties and values. The supplied properties and values are applied before each acquisition.

- `ccCustomPropertyBag customValues() const;`

  Returns a **ccCustomPropertyBag** containing the current properties and values for this FIFO.

  .

ccCustomPropertyBag

```
#include <ch_cvl/prop.h>

class ccCustomPropertyBag ;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class provides a simple container for a collection of properties, where each property is made up of a name and a value, both of type **ccCvlString**, and a type, which defines whether the item is a property that is written or a command that is executed.

This class is intended ti be a light-weight container that allows you to create a collection of properties in a single function call. You can only add to and enumerate the items in a **ccCustomPropertyBag**. To change the contents of a **ccCustomPropertyBag**, destroy it and create a new one.

Enumerations

ceType

```
enum ceType
```

An enumeration defining the property types.

Value	Meaning
ckWrite	The property is a value that is simply written.
ckCommand	The property is a command that is executed.

Constructors/Destructors

Default constructors and destructors are used.

Public Member Functions

addItem

```
void addItem(ceType type, const ccCvlString& name,
   const ccCvlString& value = cmT(""));
```

Adds the supplied item to this **ccCustomPropertyBag**. A **ccCustomPropertyBag** may contain multiple items with the same name; any existing item with the same name is retained in the collection. The new item is added as the last item of the collection.

This function does not throw any errors. If you provide an illegal or incorrectly formatted item, the acquisition system will generate an error when you attempt to acquire an image.

Parameters

type — The type of the item. *type* must be one of the following values:

ccCustomPropertyBag::ckWrite
ccCustomPropertyBag::ckCommand

name — The name of the item.

value — The value of the item. This parameter is ignored if *type* is *ccCustomPropertyBag::ckCommand*.

getItemCount

```
c_Int32 getItemCount() const;
```

Returns the number of items in the collection.

getName

```
ccCvlString getName(c_Int32 n) const;
```

Returns the name of the specified item.

Parameters

n — The index of the item. The first item has an index of 0.

Throws

ccCustomPropertyBag::BadParams
n is greater than or equal to the number of items returned by **getItemCount()**.

getValue

```
ccCvlString getValue(c_Int32 n) const;
```

Returns the value of the specified item.

Parameters

n — The index of the item. The first item has an index of 0.

Throws

ccCustomPropertyBag::BadParams

n is greater than or equal to the number of items returned by **getItemCount()**.

getType

```
eCustomPropertyType getType(c_Int32 n) const;
```

Returns the type of the specified item. The returned value is one of

ccCustomPropertyBag::ckWrite
ccCustomPropertyBag::ckCommand

Parameters

n The index of the item. The first item has an index of 0.

Throws

ccCustomPropertyBag::BadParams

n is greater than or equal to the number of items returned by **getItemCount()**.

toString

```
ccCvlString toString() const;
```

Returns a single string containing the entire contents of this **ccCustomPropertyBag**. Multiple items are separated by newline characters, and each item is formatted as follows:

type\t*name*\t*value*\n\r

where

type is “Command” or “Write”

name is the property name

value is the property value (not present for commands)

fromString

```
void fromString(ccCvlString source);
```

Replaces the current contents of this **ccCustomPropertyBag** with the values in the supplied **ccCvlString**.

If the supplied **ccCvlString** contains multiple items, they must be separated with newline characters. Individual items must be formatted as follows:

type\t*name*\t*value*\n\r

where

type is “Command” or “Write”

name is the property name

value is the property value (not present for commands)

The following is an example of a valid string:

```
Write   TestPatternSelect 1024
Write   TestPatternEnable 762
Command AutoFocus
```

Parameters

source The items to set, formatted as described above.

Throws

ccCustomPropertyBag::BadParams

string does not conform to the formatting rules described above.

Notes

You must use a DOS-style newline composed of a carriage return followed by a line feed character when formatting the string.

ccDeBoorSpline

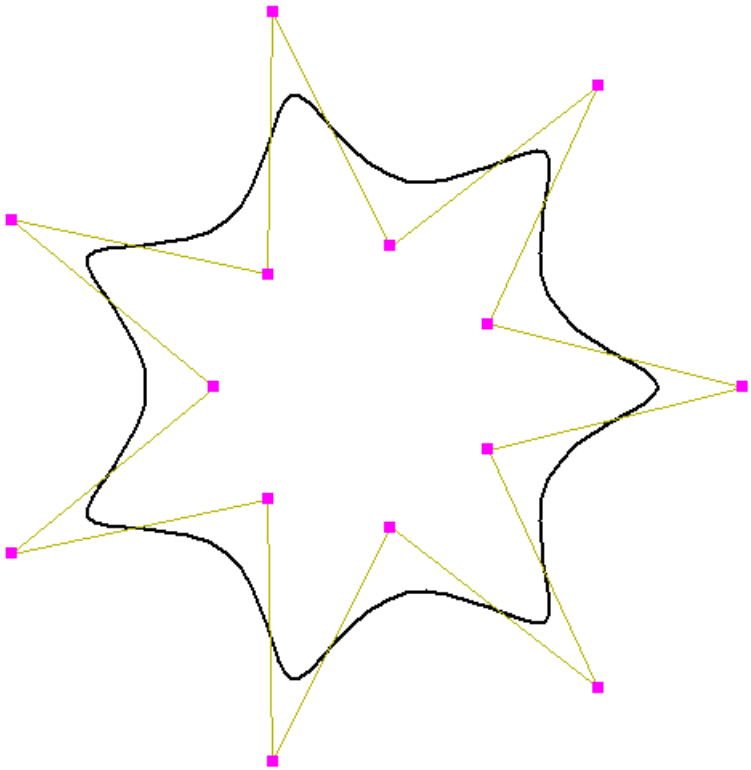
```
#include <ch_cvl/spline.h>

class ccDeBoorSpline : public ccCubicSpline;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccDeBoorSpline** class is an implementation of 2D cubic de Boor splines, a type of cubic B-Spline in which the spline control points are used to generate the Bezier control points according to the de Boor algorithm.



The above figure shows an example of a de Boor spline with control points in the shape of a star. The spline does not interpolate any of the control points.

Properties of de Boor Splines

The following properties apply to all de Boor splines:

- The curve lies within the convex hull of the spline control points.
- Moving a spline control point affects only a limited number of component Bezier curves (four if the parameterization is independent of control point geometry, six if it is not).
- Any assignment of non-negative weights to the spline control points produces a valid de Boor Spline.

The spline curve does not generally interpolate (pass through) the spline control points except for the first and last control point of an open spline.

The minimum number of spline control points needed to specify a non-empty **ccDeBoorSpline** is four for an open spline, and three for a closed spline. In other words, an open **ccDeBoorSpline** is empty if it has fewer than four control points; a closed **ccDeBoorSpline** is empty if it has fewer than three control points.

Weights

As implemented in CVL, de Boor spline control points have associated positive scalar weights, which default to unity (1.0). Increasing the weight of any control point pulls the spline curve toward that point; decreasing the weight reduces the effect of that point on the curve. Weights are relative quantities; multiplying all control point weights by a common factor has no effect on the curve. When the weights of the control points are not all equal, a de Boor spline is a form of *rational* spline. Because the parameterization may also be non-uniform, the type of de Boor splines implemented in CVL are sometimes called NURBS (Non-Uniform, Rational B-splines).

Note For in-depth information on the theory and use of 2D cubic Bezier curves and splines, see any textbook on the subject, such as *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, 1990, ISBN 0-12-249051-7.

Note All methods defined for this class leave the **ccDeBoorSpline** object unchanged when they throw an error.

Operators

operator==

```
bool operator==(const ccDeBoorSpline &rhs) const;
```

Returns true if and only if this **ccDeBoorSpline** is equal to *rhs*. Two splines are equal if all of their defining parameters are identical (no tolerance is used).

Parameters

rhs The other **ccDeBoorSpline**.

Notes

Splines that describe identical curves are not necessarily equal.

operator!=

```
bool operator!=(const ccDeBoorSpline &rhs) const;
```

Returns the opposite truth value to **operator==()**.

Parameters

rhs	The other **ccDeBoorSpline**.

Constructors/Destructors

ccDeBoorSpline

```
explicit
ccDeBoorSpline(bool isClosed = false,
  IntervalMode intervalMode = eUniform);

explicit
ccDeBoorSpline(const cmStd vector<cc2Vect> &ctrlPts,
  bool isClosed = false,
  IntervalMode intervalMode = eUniform);

ccDeBoorSpline(const cmStd vector<cc2Vect> &points,
  const cmStd vector<double> &weights,
  bool isClosed = false,
  IntervalMode intervalMode = eUniform);

explicit
ccDeBoorSpline(const ccBezierCurve &bezierCurve);
```

-
```
explicit
ccDeBoorSpline(bool isClosed = false,
  IntervalMode intervalMode = eUniform);
```

Constructs a de Boor spline with the given open/closed state and interval mode. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

isClosed	The open/closed state (default = open).
intervalMode	The interval mode. Must be one of the following: *ccCubicSpline::eUniform (default)* *ccCubicSpline:eChordLength* *ccCubicSpline::eCentripetal* *ccCubicSpline::eFixed*

- ```
  explicit
  ccDeBoorSpline(const cmStd vector<cc2Vect> &ctrlPts,
    bool isClosed = false,
    IntervalMode intervalMode = eUniform);
  ```

  Constructs a **ccDeBoorSpline** with the given control points, open/closed state, and interval mode. The weights of all control points are initialized to 1.0.

  Parameters

  ctrlPts — The vector of control points.

  isClosed — The open/closed state (default = open).

  intervalMode — The interval mode. Must be one of the following:
  ccCubicSpline::eUniform (default)
  ccCubicSpline:eChordLength
  ccCubicSpline::eCentripetal
  ccCubicSpline::eFixed

- ```
  ccDeBoorSpline(const cmStd vector<cc2Vect> &points,
    const cmStd vector<double> &weights,
    bool isClosed = false,
    IntervalMode intervalMode = eUniform);
  ```

  Constructs a **ccDeBoorSpline** with the given control points, weights, open/closed state, and interval mode.

  Parameters

  points — The vector of control points.

  weights — The vector of control point weights.

  isClosed — The open/closed state (default = open).

  intervalMode — The interval mode. Must be one of the following:
  ccCubicSpline::eUniform (default)
  ccCubicSpline:eChordLength
  ccCubicSpline::eCentripetal
  ccCubicSpline::eFixed

  Throws

  ccShapesError::BadParams
  points and *weights* vectors have different sizes.

  ccShapesError::NonpositiveWeight
  Any weight is non-positive.

- `explicit ccDeBoorSpline(const ccBezierCurve &bezierCurve);`

  Constructs a de Boor spline that describes the same geometry as the specified Bezier curve.

  Parameters

  bezierCurve The Bezier curve.

  Notes

  The constructed spline is open, has four control points, has a single Bezier curve that is equal to the supplied Bezier curve, and has a **maxParam()** value of 1.0.

Public Member Functions

weight

```
double weight(c_Int32 idx) const;

void weight(c_Int32 idx, double newWeight);
```

- `double weight(c_Int32 idx) const;`

  Gets the weight of the control point with the given index.

  Parameters

  idx The index.

  Throws

  ccShapesError::BadIndex

  idx is less than 0, or greater than or equal to **numControlPoints()**.

- `void weight(c_Int32 idx, double newWeight);`

  Sets the weight of the control point with given index.

  Parameters

  idx The index.

  newWeight The new weight.

  Throws

  ccShapesError::BadIndex

  idx is less than 0, or greater than or equal to **numControlPoints()**.

ccShapesError::NonpositiveWeight
newWeight is non-positive.

weights

```
const cmStd vector<double> &weights() const;

void weights(const cmStd vector<double> &wghts);
```

- `const cmStd vector<double> &weights() const;`

  Gets the vector of control point weights.

- `void weights(const cmStd vector<double> &wghts);`

  Sets the vector of control point weights.

  Parameters
  wghts The vector of control point weights.

  Throws
  ccShapesError::BadParams
  The size of *wghts* is not equal to **numControlPoints()**.

  ccShapesError::NonpositiveWeight
  Any element of *wghts* is non-positive.

insertControlPoint

```
void insertControlPoint(c_Int32 idx, const cc2Vect &point,
 double weight);

virtual void insertControlPoint(c_Int32 idx,
 const cc2Vect &point);
```

- `void insertControlPoint(c_Int32 idx, const cc2Vect &point, double weight);`

  Identical to **ccCubicSpline::insertControlPoint()**, except that the new control point is assigned the given weight rather than the default weight of 1.0.

  Parameters
  idx The index.

  weight The weight.

Throws

ccShapesError::BadIndex
idx is less than 0, or greater than **numControlPoints()** before the insertion.

ccShapesError::NonpositiveWeight
weight is non-positive.

See **ccCubicSpline::insertControlPoint()** for more information.

- ```
  virtual void insertControlPoint(c_Int32 idx,
    const cc2Vect &point);
  ```

Inserts a new control point before the point with the given index.

Parameters

idx The index.

point The control point.

Throws

ccShapesError::BadIndex
idx is less than 0, or greater than **numControlPoints()** before the insertion.

Notes

The new control point is assigned a weight of 1.0.

See **ccCubicSpline::insertControlPoint()** for more information.

addControlPoint

```
void addControlPoint(const cc2Vect &point,
  double weight = 1.0);
```

Convenience function. Appends a control point at the end of the sequence of control points. Equivalent to **ccCubicSpline::insertControlPoint(numControlPoints(), point, weight)**.

Parameters

point The control point.

weight The weight (default = 1.0).

Throws

ccShapesError::NonpositiveWeight
weight is non-positive.

See **ccCubicSpline::addControlPoint()** for more information.
```

removeControlPoint

```
virtual void removeControlPoint(c_Int32 idx);
```

Removes the control point with the given index.

Parameters

idx The index.

Throws

ccShapesError::BadIndex

idx is less than 0, or greater than **numControlPoints()** before the insertion.

See **ccCubicSpline:removeControlPoint()** for more information.

map

```
ccDeBoorSpline map(const cc2Xform& X) const;
```

Returns this **ccDeBoorSpline** mapped by *X*.

Parameters

X The transformation object.

Notes

This function maps only control point positions by *X* while leaving weights and intervals unchanged. It may be desirable to invoke **reparameterize()** on the returned spline. See *Mapping Splines* and *Reparameterizing Splines* on page 1112 for more information.

Note Several of **ccCubicSpline**'s setter/getter pairs have a virtual setter and a non-virtual getter. The getters must be explicitly exposed in derived classes, or they will be hidden by the setter overrides.

isClosed

```
bool isClosed() const;

virtual void isClosed(bool);
```

- `bool isClosed() const;`

  Gets the open/closed state of this **ccDeBoorSpline** (true = closed, false = open).

- `virtual void isClosed(bool);`

  Sets the open/closed state of this **ccDeBoorSpline**.

  Parameters

  bool True sets this spline to closed, false sets it to open.

See **ccControlPoint::isClosed()** for more information.

controlPoint

```
cc2Vect controlPoint(c_Int32 idx) const;

virtual void controlPoint(c_Int32 idx,
  const cc2Vect &ctrlPt);
```

- `cc2Vect controlPoint(c_Int32 idx) const;`

  Gets the control point with the given index.

  Parameters

  idx The index.

- `virtual void controlPoint(c_Int32 idx, const cc2Vect &ctrlPt);`

  Sets the control point with the given index.

  Parameters

  idx The index.

  ctrlPt The control point.

  Notes

  This function leaves the weight of the control point unchanged.

  See **ccCubicSpline::controlPoint()** for more information.

controlPoints

```
const cmStd vector<cc2Vect> &controlPoints() const;

virtual void controlPoints(
  const cmStd vector<cc2Vect> &ctrlPts);
```

- `const cmStd vector<cc2Vect> &controlPoints() const;`

  Gets the entire vector of control points.

- `virtual void controlPoints( const cmStd vector<cc2Vect> &ctrlPts);`

  Sets the entire vector of control points.

  Parameters

  ctrlPts The vector of control points.

Notes

This function leaves the weights of existing control points unchanged. If *ctrlPts* has size *n*, the weights of the first *n* existing control points are used for the new points. If there are fewer than *n* existing control points, the surplus points in *ctrlPts* receive weights of 1.0.

See **ccCubicSpline::controlPoints()** for more information.

reparameterize

```
virtual void reparameterize();
```

Recomputes all intervals based on the current control points and interval mode. See **ccCubicSpline::reparameterize()** for more information.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this de Boor spline.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this de Boor spline. See **ccShape::reverse()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this **ccDeBoorSpline** mapped by *X*.

Parameters

X	The transformation object.

See **ccShape::mapShape()** for more information.

ccDegree

```
#include <ch_cvl/units.h>

class ccDegree;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes an angle in degrees.

Note that you can use the constructors to convert from one angle representation to another. For example, to specify π radians in degrees, you could write:

```
 // ckPI is defined in <ch_cvl/math.h>
ccDegree deg(ccRadian(ckPI));
```

Constructors/Destructors

ccDegree

```
ccDegree();

ccDegree(ccRadian a);

ccDegree(ccAngle8 a);

ccDegree(ccAngle16 a);

explicit ccDegree(double a);
```

- `ccDegree();`

  Creates an uninitialized **ccDegree** object.

- `ccDegree(ccRadian a);`

  Creates a **ccDegree** object from the given **ccRadian** object.

  Parameters

a	The **ccRadian** object to convert to degrees.

- `ccDegree(ccAngle8 a);`

  Creates a **ccDegree** object from the given **ccAngle8** object.

  Parameters

  a The **ccAngle8** object to convert to degrees.

- `ccDegree(ccAngle16 a);`

  Creates a **ccDegree** object from the given **ccAngle16** object.

  Parameters

  a The **ccAngle16** object to convert to degrees.

- `explicit ccDegree(double a);`

  Creates a **ccDegree** object with the specified value.

  Parameters

  a The angle in degrees.

Operators

operator+

```
ccDegree operator+(ccDegree a) const;
```

Returns the result of adding the angle *a* in degrees to this angle.

Parameters

a The angle to add to this angle.

operator-

```
ccDegree operator-(ccDegree a) const;
ccDegree operator-() const;
```

- `ccDegree operator-(ccDegree a) const;`

  Returns the result of subtracting the angle *a* in degrees from this angle.

  Parameters

  a The angle to subtract from this angle.

- `ccDegree operator-() const;`

  Returns the negative of this angle. The unary minus operator.

operator*

```
ccDegree operator*(double a) const;
double operator*(ccDegree a) const;
friend ccDegree operator*(double a, ccDegree b);
```

- `ccDegree operator*(double a) const;`

  Returns the result of multiplying this angle by *a*.

  Parameters

  a The amount to multiply by.

- `double operator*(ccDegree a) const;`

  Returns the result of multiplying this angle by the angle *a* in degrees.

  Parameters

  a The angle to multiply by.

- `friend ccDegree operator*(double a, ccDegree b);`

  Returns the result of multiplying the angle *b* by *a*.

  Parameters

  a The amount to multiply by.

  b The angle to multiply.

operator/

```
ccDegree operator/(double a) const;
double operator/(ccDegree a) const;
```

- `ccDegree operator/(double a) const;`

  Returns the result of dividing this angle by *a*.

  Parameters

  a The amount to divide by.

- `double operator/(ccDegree a) const;`

  Returns the result of dividing this angle by the angle *a* in degrees.

  Parameters

  a The angle to divide by.

operator=

```
ccDegree& operator=(double a);
```

Assigns the value *a* to this angle.

Parameters

a The value to assign.

---

operator*=

```
ccDegree& operator*=(double a);
ccDegree& operator*=(ccDegree a);
```

- ```
  ccDegree& operator*=(double a);
  ```

  Multiplies this angle by *a* and returns the result.

  Parameters

  a The value to multiply by.

- ```
  ccDegree& operator*=(ccDegree a);
  ```

  Multiplies this angle by the angle *a* and returns the result.

  Parameters

  a The angle to multiply by.

---

operator/=

```
ccDegree& operator/=(double a);
ccDegree& operator/=(ccDegree a);
```

- ```
  ccDegree& operator/=(double a);
  ```

  Divides this angle by the value *a* and returns the result.

  Parameters

  a The value to divide by.

- ```
  ccDegree& operator/=(ccDegree a);
  ```

  Divides this angle by the angle *a* and returns the result.

  Parameters

  a The angle to divide by.

operator+= `ccDegree& operator+=(ccDegree a);`

Adds the angle *a* to this angle and returns the result.

Parameters

a	The angle to add.

operator-= `ccDegree& operator-=(ccDegree a);`

Subtracts the angle *a* from this angle and returns the result.

Parameters

a	The angle to subtract.

operator== `bool operator!=(ccDegree a) const;`

Returns true if this angle is exactly equal to the angle *a*.

Parameters

a	The other angle.

operator!= `bool operator!=(ccDegree a) const;`

Returns true if this angle is not equal to the angle *a*.

Parameters

a	The other angle.

operator< `bool operator<(ccDegree a) const;`

Returns true if this angle is less than angle *a*.

Parameters

a	The other angle.

operator<= `bool operator<=(ccDegree a) const;`

Returns true if this angle is less than or equal to angle *a*.

Parameters

a	The other angle.

operator> `bool operator>(ccDegree a) const;`

Returns true if this angle is greater than angle *a*.

Parameters

a The other angle.

operator>= `bool operator>(ccDegree a) const;`

Returns true if this angle is greater than or equal to angle *a*.

Parameters

a The other angle.

Public Member Functions

toDouble `double toDouble() const;`

Returns this angle as a **double** value.

plain `double plain() const;`

Returns this angle as a **double**.

norm `ccDegree norm() const;`

Returns this angle normalized to the range from 0 up to (but not including) 360.0.

signedNorm `ccDegree signedNorm() const;`

Returns this angle normalized to the range from -180.0 up to (but not including) 180.0.

ccDiagDefs

```
#include <ch_cvl/diagobj.h>
```

```
class ccDiagDefs;
```

A name space class that holds enumerations used with tool diagnostics.

Enumerations

What

```
enum What;
```

What tool operations to record.

Value	Meaning
eInputs	Record tool inputs.
eIntermediate	Record tool intermediate results.
eResults	Record tool final results.

When

```
enum When;
```

Flag to record tool operations.

Value	Meaning
eRecordOn	Recording enabled
eRecordOff	Recording disabled

```
enum {eShallowCopy = 0x400000};
```

Value	Meaning
eShallowCopy	Record all handled data with shallow copies. The default is to record tool results with deep copies.

```
enum {eRecordDefault = eInputs | eResults | eRecordOn};
```

Value	Meaning
eRecordDefault	Record inputs and results.

ccDiagDefs

ccDiagIncrementLevel

```
#include <ch_cvl/diagobj.h>

class ccDiagIncrementLevel;
```

This is a helper class for vision tool implementers.

Constructors/Destructors

ccDiagIncrementLevel

```
ccDiagIncrementLevel(ccDiagObject* obj);

~ccDiagIncrementLevel();
```

- `ccDiagIncrementLevel(ccDiagObject* obj);`

  Increments the level of the given diagnostic object.

  Parameters

  obj The diagnostic object.

  Notes

  When *obj* is 0, the object has no effect. *obj* must not be deleted until *this* object has been destroyed.

`~ccDiagIncrementLevel();`

Decrements the level of the given diagnostic object.

ccDiagIncrementLevel

ccDiagObject

```
#include <ch_cvl/diagobj.h>

class ccDiagObject;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This is a container class to hold **ccDiagRecord** objects.

Constructors/Destructors

ccDiagObject

```
ccDiagObject();
```

Creates a diagnostic object with a level of zero and no diagnostic records. See **level()** on page 1155.

Public Member Functions

level

```
c_Int32 level() const;

void level(c_Int32 lvl);
```

- `c_Int32 level() const;`

  Gets the current level for new record creation.

- `void level(c_Int32 lvl);`

  Sets the level for new record creation. The level of the record is the nested depth of the tool which created it. The top-level tool adds records with a level of zero. As each tool calls sub-tools, the sub-tools create records with increasing level values.

  Parameters

  lvl The level to be used for new record creation.

  Throws

  ccDiagDefs::InvalidLevel The given level is less than zero.

newDiagRecord

```
ccDiagRecord& newDiagRecord();
```

Allocates a new diagnostic record and returns a reference to the newly created record. The new record has its level set to **level()**. The returned reference should not be cached for later use. See also **reset()** on page 1156.

Notes

All references to records in this object previously returned by **newDiagRecord()** or **diagRecord()** are invalid after calling this function.

numRecords

```
c_Int32 numRecords() const;
```

Returns the total number of records in this object.

diagRecord

```
ccDiagRecord& diagRecord(c_Int32 num);
```

Retrieves a previously allocated record. This reference should not be cached. See also **newDiagRecord()** on page 1156 and **reset()** on page 1156.

Parameters

num	The number of the record to retrieve. Records are numbered from 0 to **numRecords()** - 1.

Throws

ccDiagDefs::InvalidRecordNumber The number is not a valid record.

reset

```
void reset();
```

Resets the diagnostic object to initialized state. All records associated with this diagnostic object are released.

Notes

All references to records in this object previously returned by **newDiagRecord()** or **diagRecord()** are invalid after calling this function.

Static Functions

setThreadRecording

```
static void setThreadRecording(ccDiagObject* obj,
  c_UInt32 flags);
```

Sets the diagnostic object used for recording all tools executed by this thread.

Parameters

obj — The diagnostic object to use for recording. If *obj* is zero, thread recording is disabled for this thread.

flags — Recording flags. Flags can be the OR of any of the **ccDiagDefs::What** values ORed with one of the **ccDiagDefs::When** values.
What values:
ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

When values:
ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

getThreadRecording

```
static ccDiagObject* getThreadRecording(c_UInt32& flags);
```

Gets the diagnostic object used for recording all tools executed by this thread.

Parameters

flags — Recording flags. Flags can be the OR of any of the **ccDiagDefs::What** values ORed with one of the **ccDiagDefs::When** values.
What values:
ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

When values:
ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

shouldRecord

```
static bool shouldRecord(c_UInt32 diagFlags,
  ccDiagDefs::What what);
```

Returns true if the given type of diagnostic data should be recorded to this diagnostic object. This function is generally only called by vision tool implementors.

Parameters

diagFlags — Current value of diagnostic recording flags.

what — Recording flag. Must be one or more of the following:
ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

getRecordingDestination

```
static ccDiagObject* getRecordingDestination(
  ccDiagObject* obj, c_UInt32& flags);
```

Takes the diagnostic object and flags passed to a vision tool run function and returns the diagnostic object and flags to be used for recording, either zero, the one passed in, or the thread recording settings. This function is generally only called by vision tool implementors.

Parameters

obj — The current diag object. If *obj* is non-zero, it overrides any thread recording object. If *obj* is zero, the thread recording diagnostic object is returned (this defaults to a null pointer).

flags — Recording flags. If *flags* is zero, the thread recording flags are returned (these default to zero). If non-zero *flags* are passed to a vision tool run function, they override any thread recording settings.
Flags can be the OR of any of the **ccDiagDefs::What** values ORed with one of the **ccDiagDefs::When** values.
What values:
ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

When values:
ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

When a given tool has data to record, it should use this function to determine which diagnostic object to record to and what flags to use.

ccDiagRecord

```
#include <ch_cvl/diagobj.h>

class ccDiagRecord;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This class holds diagnostic record objects.

Enumerations

ItemType

```
enum ItemType;
```

This enumeration defines the type of an item in the **ccDiagRecord**.

Value	Meaning
eGraphics	Item is a graphic object.
eText	Item is a text object.
eData	Item is a data object.
eSketch	Item is a sketch object.

Notes

eSketch is provided for backward compatibility only. *eGraphics* should be used for all new application code.

Constructors/Destructors

ccDiagRecord

```
ccDiagRecord(c_Int32 level=0);
```

Constructs a diagnostic record with a level of 0 and no items. See comments for member function **level()** on page 1160.

Parameters

level The level of the record.

Throws
ccDiagDefs::InvalidLevel
If *level* is < 0.

Public Member Functions

level

```
c_Int32 level() const;
void level(c_Int32 lvl);
```

- `c_Int32 level() const;`

  Gets the level of the record

- `void level(c_Int32 lvl);`

  Sets the level of this record. The level of a record reflects the nested depth of the tool which created it. The top-level tool adds records with a level of zero. As each tool calls sub-tools, the sub-tools create records with increasing level values.

  Parameters
  lvl The level of the record

  Throws
  ccDiagDefs::InvalidLevel
  If given level is < 0.

recordAnnotation

```
ccCvlString recordAnnotation() const;
void recordAnnotation(const ccCvlString& annot);
```

- `ccCvlString recordAnnotation() const;`

  Gets the annotation string for this record.

- `void recordAnnotation(const ccCvlString& annot);`

  Sets the annotation string for this record.

  Parameters
  annot The annotation string.

Notes

Each record has an annotation string, and each item in a record has its own annotation string.

---

image

```
ccPelBuffer_const<c_UInt8>& image() const;

void image(const ccPelBuffer_const<c_UInt8>& pb,
  bool allowOverwrite = false);
```

- `ccPelBuffer_const<c_UInt8>& image() const;`

Returns the image for this record. If no image has been set, returns an unbound pel buffer.

- `void image(const ccPelBuffer_const<c_UInt8>& pb, bool allowOverwrite = false);`

Adds an image to the current record.

Parameters

pb The image.

allowOverwrite If *allowOverwrite* is true, the given pel buffer will replace the current pel buffer. If *allowOverwrite* is false, attempting to add more than one pel buffer to a record will result in a throw.

Throws

ccDiagDefs::ImageItemNotAvailable
When attempting to add more than one image and *allowOverwrite* == **true**.

Notes

A record can have only one image. The pel buffer image must not be modified until the **ccDiagObject** is no longer using it.

numItems

```
c_Int32 numItems() const;
```

Returns the number of items in this record.

itemType

```
ItemType itemType(c_Int32 num) const;
```

Gets the type of the given item. Items are numbered from 0 to **numItems()** - 1.

Parameters

num The number of the item.

Throws

ccDiagDefs::InvalidItemNumber
If *num* is < 0 or >= **numItems()**

graphics

```
void graphics(c_Int32 num, ccGraphicList& graphics,
 ccDisplay::CoordinateSystem& coord,
 ccCvlString& itemAnnot) const;

void graphics(const ccGraphicList& graphics,
 ccDisplay::CoordinateSystem coord,
 const ccCvlString& itemAnnot);
```

- ```
  void graphics(c_Int32 num, ccGraphicList& graphics,
   ccDisplay::CoordinateSystem& coord,
   ccCvlString& itemAnnot) const;
  ```

  Returns the data associated with the given *eGraphics* item.

  Parameters

num	The item number to be retrieved.
graphics	The graphics list for the given item.
coord	The graphics list coordinate system for the given item.
itemAnnot	The annotation string for the given item.

  Throws

  ccDiagDefs::InvalidItemNumber
  If *num* < 0 or >= **numItems().**

  ccDiagDefs::InvalidItemType
  If the given item is not of type *eGraphics.*

- ```
  void graphics(const ccGraphicList& graphics,
   ccDisplay::CoordinateSystem coord,
   const ccCvlString& itemAnnot);
  ```

  Appends a new *eGraphics* item with the given data to the current list of items.

  Parameters

graphics	The graphics list for the new item.
coord	The graphics coordinate system for the new item.
itemAnnot	The annotation string for the new item.

text

```
void text(c_Int32 num, ccCvlString& text,
 ccCvlString& itemAnnot) const;

void text(const ccCvlString& text,
 const ccCvlString& itemAnnot);
```

- ```
  void text(c_Int32 num, ccCvlString& text,
   ccCvlString& itemAnnot) const;
  ```

  Returns the data associated with the given *eText* item.

  Parameters

  num The item number to be retrieved.

  text The text for the given item.

  itemAnnot The annotation string for the given item.

  Throws

  ccDiagDefs::InvalidItemNumber
  If *num* < 0 or >= **numItems().**

  ccDiagDefs::InvalidItemType
  If the given item is not of type *eText*.

- ```
  void text(const ccCvlString& text,
   const ccCvlString& itemAnnot);
  ```

  Appends a new *eText* item with the given data to the current list of items.

  Parameters

  text The text string for the new item.

  itemAnnot The annotation string for the new item.

reset

```
void reset();
```

Reset the record. Clears all images, text, graphics, and data, and resets the record annotation and level.

Deprecated Members

sketch

```
void sketch(c_Int32 num, ccUISketch& sketch,
 ccDisplay::CoordinateSystem& coord,
 ccCvlString& itemAnnot) const;

void sketch(const ccUISketch& graphics,
 ccDisplay::CoordinateSystem coord,
 const ccCvlString& itemAnnot);
```

- ```
  void sketch(c_Int32 num, ccUISketch& sketch,
   ccDisplay::CoordinateSystem& coord,
   ccCvlString& itemAnnot) const;
  ```

  Returns the data associated with the given *eSketch* item.

  Parameters

  num Item number to be retrieved.

  sketch Sketch for the given item.

  coord Sketch coordinate system for the given item.

  itemAnnot Annotation string for the given item.

  Throws

  *ccDiagDefs::*InvalidItemNumber
  If num < 0 or >= **numItems().**

  *ccDiagDefs::*InvalidItemType
  Given item is not of type *eSketch.*

- ```
  void sketch(const ccUISketch& sketch,
  ccDisplay::CoordinateSystem coord, const ccCvlString&
  itemAnnot);
  ```

  Appends a new *eSketch* item with the given data to the current list of items.

  Parameters

  sketch Sketch for the given item.

  coord Sketch coordinate system for the given item.

  itemAnnot Annotation string for the given item.

ccDiagServer

```
#include <ch_cvl/diagsrv.h>

class ccDiagServer;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is a server for **ccDiagObject** objects.

Constructors/Destructors

Constructors and destructors are automatically called by the program for this class.

Static Functions

init

```
static void init();
```

Initializes the diagnostics server. This function must be called before trying to show any diagnostic objects using **ccDiagServer:: showDiagObject()**. If this function is called, do not call **ccRPC::init()**, since this is done within this function.

The host application must call **ccDiagServer::init()** at system initialization and **ccDiagServer::exit()** at system shutdown if diagnostic objects are going to be displayed on the host, whether the call to**ccDiagServer:: showDiagObject()** is executed on the host or on the embedded side. If an embedded application is going to call **ccDiagServer::showDiagObject()** during its lifetime, it also must call the **ccDiagServer::init()** and **ccDiagServer::exit()** function pair.

Notes

ccDiagServer::init() must be called only once at system startup. Attempting to call this function more than once will result in an error.

exit

```
static void exit();
```

Shuts down all threads and releases all resources used by the display server for all accelerators.

Notes

ccDiagServer::exit() must be called only once at system shutdown. Attempting to call this function more than once will result in an error.

showDiagObject

```
static void showDiagObject(ccAccelerator* acc,
  const c_Int32 channel, ccDiagObject& diagObject);
```

Displays a diagnostic object to the specified channel (window) associated with the specified accelerator.

Parameters

acc	The accelerator, 0 specifies the host.
channel	The channel number (window) on the host.
diagObject	The diagnostic object to be displayed.

ccDIB

```
#include <ch_cvl/dib.h>

class ccDIB;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Not applicable

The **ccDIB** class lets you convert Windows DIB (Device-Independent Bitmap) files into CVL **ccPelBuffer**s and CVL **ccPelBuffer**s into Windows DIB files.

Constructors/Destructors

ccDIB

```
ccDIB();
```

Constructs an uninitialized **ccDIB** object with no image pixel or palette data. Member functions that require an initialized **ccDIB** object will throw if it is uninitialized.

Public Member Functions

init

```
void init(const ccCvlString& filename);
void init(ccCvlIStream& stream);
void init(const ccPelBuffer_const<c_UInt8>& pelBuf);
void init(const c_UInt8* dib, c_UInt32 len);
```

- `void init(const ccCvlString& filename);`

Reads a DIB from the supplied file and initializes this object with pixel and palette data.

Parameters

filename The file from which to read the DIB.

Throws

ccDIB::ReadError
The supplied file cannot be opened or read.

ccDIB::BadFormat
The supplied file is not in the proper DIB format.

- `void init(ccCvlIStream& stream);`

Reads a DIB from the supplied **ccCvlIStream** and initializes this object with pixel and palette data.

Parameters
stream The **ccCvlIStream** from which to read the DIB.

Throws
ccDIB::BadFormat
The supplied **ccCvlIStream** is not in the proper DIB format.

- `void init(const ccPelBuffer_const<c_UInt8>& pelBuf);`

Given a **ccPelBuffer**, initializes this object with pixel data from the **ccPelBuffer** and a 256 color grey-scale palette.

Parameters
pelBuf The pixel data used to initialize this object.

- `void init(const c_UInt8* dib, c_UInt32 len);`

Given a DIB in packed-memory DIB format, initializes this **ccDIB** with pixel and palette data from the DIB.

Parameters
dib A pointer to a DIB in packed-memory format.

len The number of bytes pointed to by *dib*.

Throws
ccDIB::BadFormat
dib does not point to a valid packed-memory format DIB.

---

write

```
void write(const ccCvlString& filename) const;
void write(ccCvlOStream& stream) const;
```

- `void write(const ccCvlString& filename) const;`

Writes this **ccDIB** object out to the specified file in DIB format.

Parameters

filename The file to which to write the DIB.

Throws

ccDIB::WriteError

file cannot be written.

ccDIB::NoDibData

This **ccDIB** was default constructed.

- `void write(ccCvlOStream& stream) const;`

Writes this **ccDIB** object out to the supplied **ccCvlOstream** in DIB format.

Parameters

stream The **ccCvlOstream** to which to write the DIB.

Throws

ccDIB::WriteError

The supplied **ccCvlOstream** cannot be written.

ccDIB::NoDibData

This **ccDIB** was default constructed.

pelBuffer `ccPelBuffer<c_UInt8> pelBuffer() const;`

Returns a **ccPelBuffer** representation of this **ccDIB** object. A color DIB is converted to a grey-scale **ccPelBuffer**. The algorithm used for converting color DIBs to 8 bit images is as follows:

$$p = (R + G + B) / 3$$

where

p is the pixel value in the grey-scale **ccPelBuffer**

R is the red value of the corresponding DIB pixel

G is the green value of the corresponding DIB pixel

B is the blue value of the corresponding DIB pixel

Throws

ccDIB::NoDibData

This **ccDIB** was default constructed.

clipboardDibSize

```
c_UInt32 clipboardDibSize() const;
```

Returns the number of bytes required to store the current contents of this **ccDIB** in packed-memory DIB format.

renderClipboardDib

```
void renderClipboardDib(c_UInt8* dest) const;
```

Copies the current contents of this **ccDIB** in packed-memory DIB format to the supplied block of memory.

Parameters

dest — The memory to which the packed-memory format DIB is copied. Note that you must allocate *dest* to point to at least the number of bytes returned by **clipboardDibSize()**.

Throws

ccDIB::NoDibData — This **ccDIB** was default constructed.

ccDigitalCameraControlProp

```
#include <ch_cvl/prop.h>

class ccDigitalCameraControlProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class allows you to control properties of certain digital cameras on platforms that support this property.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccDigitalCameraControlProp

```
ccDigitalCameraControlProp();
```

Creates a digital camera property not associated with any acquisition FIFO. By default, high gain is off.

Constants

defaultMasterClockFrequency

```
static const double defaultMasterClockFrequency;
```

The default master clock frequency: 32.0 MHz.

Public Member Functions

selectHighGain

```
void selectHighGain(bool isHigh);

bool selectHighGain() const;
```

- `void selectHighGain(bool isHigh);`

  Sets the digital camera's analog-to-digital (A/D) gain either high (true) or low (false).

  Some digital cameras provide only manual gain control by means of a physical switch or a dial in the camera. However, some digital cameras, such as the Dalsa Spyder line scan cameras, support gain selection by means of a control line.

  Parameters

  isHigh — True to set the A/D gain adjustment high; false to set it low.

- `bool selectHighGain() const;`

  Returns whether the digital camera's analog-to-digital (A/D) gain is set high (true) or low (false). Not all digital cameras support this feature.

masterClockFrequency

```
void masterClockFrequency(double frequency);

double masterClockFrequency() const;
```

- `void masterClockFrequency(double frequency);`

  Sets the master clock frequency in megahertz.

  Parameters

  frequency — The master clock frequency in MHz.

  Throws

  ccDigitalCameraControlProp::BadParams — The frequency is not in the range 5.0 to 40.0.

- `double masterClockFrequency() const;`

  Returns the master clock frequency in megahertz.

ccDimTol

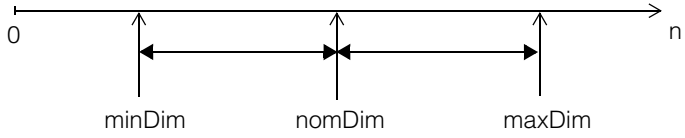
```
#include <ch_cvl/dimtol.h>

class ccDimTol;
```

Class Properties

Copyable	Yes
Derivable	Not intended
Archiveable	Simple

This class embodies information about dimension tolerances. Specifically, a nominal dimension, and its minimum and maximum values.



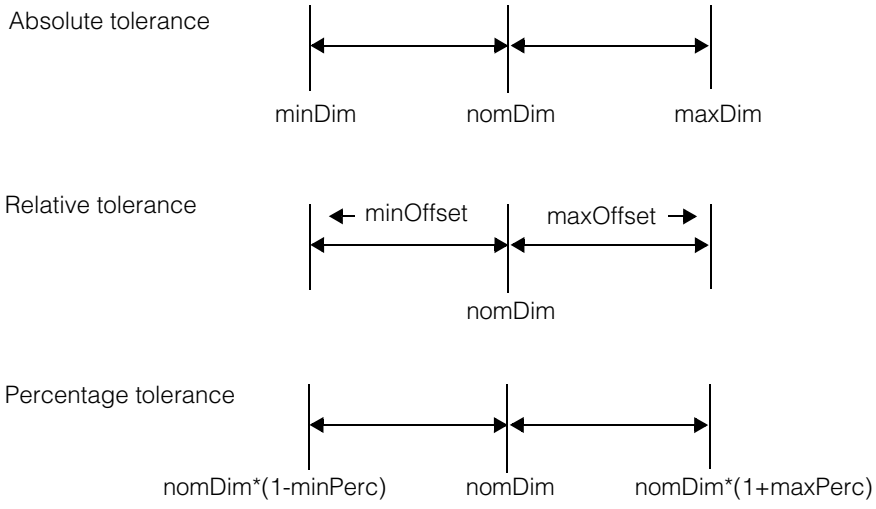
The class allows a dimension minimum and maximum tolerance to be specified using one of three measurement types:

- As percentages of the nominal dimension
- As relative offsets from the nominal dimension
- As absolute limits on the dimension.

In this manner, it is possible to store tolerances without explicit knowledge of the nominal dimension. If the nominal dimension is known, however, the class provides methods for converting tolerance information from one measurement type to another.

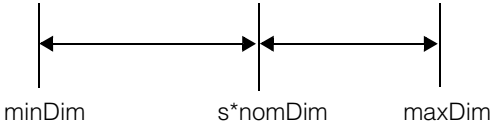
Note that both minimum and maximum tolerances are always specified as positive numbers (relative offsets and percentages add and subtract from the nominal). Furthermore, the minimum tolerance is constrained such that the absolute minimum dimension can never be less than zero. However, either the maximum or minimum tolerances can be specified as *undefined* using a value of -1, which is the default.

The three measurement types are illustrated below:

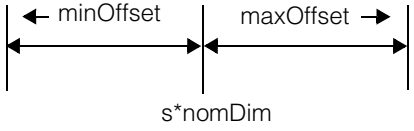


When you apply a scale factor to a **ccDimTol** object, the effect depends on the tolerance type as illustrated below. (scale factor = s)

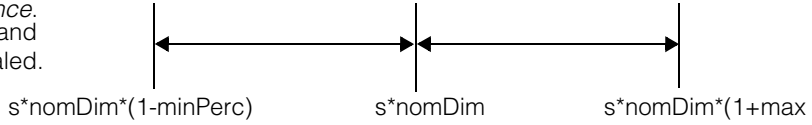
Absolute tolerance.
nomDim scales but the min and max limits stay fixed.



Relative tolerance.
nomDim, minDim, and maxDim scale, but the offsets remain fixed



Percentage tolerance.
nomDim, minDim, and maxDim are all scaled.



Constructors/Destructors

ccDimTol

```
ccDimTol(
  double nomDim = -1,
  double minTol = -1,
  double maxTol = -1,
  TolType type = eAbsolute);
```

Constructs an object with the specified absolute dimension tolerances, and the tolerance type *eAbsolute*. The tolerance type dictates how the *minTol* and *maxTol* values you specify are interpreted. See the reference page introductory information.

Parameters

nomDim	The nominal dimension.
minTol	The minimum tolerance.
maxTol	The maximum tolerance.
type	The tolerance type.

If your constructor specifies *type=eAbsolute*, the following throws apply.

Throws

ccShapesError::BadGeom
If *minDim* < 0 and not -1,
or if *maxDim* < 0 and not -1,
or if *minDim* and *maxDim* are defined and *minDim* > **maxTol()**,
or if *minDim* and *nomDim* are defined and *minDim* > *nomDim*,
or if *maxDim* and *nomDim* are defined and *maxDim* < *nomDim*.

If your constructor specifies *type=eRelative*, the following throws apply.

Throws

ccShapesError::BadGeom
If *minOffset* < 0 and not -1,
or if *maxOffset* < 0 and not -1,
or if *minOffset* and *nomDim* are defined and *minOffset* > *nomDim*.

If your constructor specifies *type=ePercentage*, the following throws apply.

Throws

ccShapesError::BadGeom
If *minPerc* < 0 and not -1,
or if *minPerc* > 1 and not -1,
or if *maxPerc* < 0 and not -1.

Enumerations

TolType

```
enum TolType
```

Value	Meaning
eAbsolute	Keep absolute tolerance limits constant.
eRelative	Keep relative tolerance offsets constants.
ePercentage	Keep tolerance percentages constant.

Operators

operator*

```
friend ccDimTol operator* (
  double scale,
  const ccDimTol& dim);
```

Scales the dimensional tolerance object *dim*. The effect of scaling on the dimension tolerances depends on the *dim* tolerance type. See the introductory information for this reference page.

Parameters

scale	Scale factor.
dim	Object to be scaled.

operator==

```
bool operator== (const ccDimTol& tol) const;
```

This operator allows you to compare two **ccDimTol** objects. For example,

```
if(dim1 == dim2) {.....}
```

The equality expression evaluates to *true* if the objects are equal. It is *false* if they are not equal.

Parameters

tol	The right-hand **ccDimTol** object. For example, ***dim2***.

operator!=

```
bool operator!= (const ccDimTol& tol) const;
```

This operator allows you to compare two **ccDimTol** objects. For example,

```
if(dim1 != dim2) {.....}
```

The inequality expression evaluates to *true* if the objects are not equal. It is *false* if they are equal.

Parameters

tol The right-hand **ccDimTol** object. For example, *dim2*.

Public Member Functions

nomDim

```
void nomDim(double d);
```

```
double nomDim() const;
```

The nominal dimension value in physical units. A -1 indicates the dimension is undefined.

- `void nomDim(double d);`

  Sets a new nominal dimension value.

  Parameters

  d The new nominal dimension value.

- `double nomDim() const;`

  Returns the nominal dimension.

  Throws

  ccShapesError::BadGeom

  If *d* <= 0 and not -1,
  or if **tolType()** = *eRelative* and **minOffset()** and *d* are defined and **minOffset()** > *d*.
  or if **tolType()** = *eAbsolute* and one of the following is true:
  minDim() and *d* are defined and *d* < **minDim()**,
  or **maxDim()** and *d* are defined and *d* > **maxDim()**.

minTol

```
double minTol() const;
```

Returns the minimum dimension tolerance. You will need to know the **tolType()** value to interpret this result.

maxTol

```
double maxTol() const;
```

Returns the maximum dimension tolerance. You will need to know the **tolType()** value to interpret this result.

tolType

```
TolType tolType() const;
void tolType(TolType type);
```

- `TolType tolType() const;`

  Returns the tolerance type.

- `void tolType(TolType type);`

  Sets a new tolerance type. Must be *eAbsolute*, *eRelative*, or *ePercentage*.

  Parameters

  type The new tolerance type.

  Throws

  ccShapesError::BadGeom
  If *type* != **tolType()** and **nomDim()** = -1. (Measurement conversion is impossible).

minDim

```
double minDim() const;
```

Returns the absolute minimum dimension tolerance. A tolerance value of -1 indicates the tolerance value is undefined.

Throws

ccShapesError::BadGeom
If the minimum dimension value is defined and **nomDim()** is not defined. (Measurement conversion is impossible).

maxDim

```
double maxDim() const;
```

Returns the absolute maximum dimension tolerance. A tolerance value of -1 indicates the tolerance value is undefined.

Throws

ccShapesError::BadGeom
If the maximum dimension value is defined and **nomDim()** is not defined. (Measurement conversion is impossible).

minOffset

```
double minOffset() const;
```

Returns the relative minimum dimension tolerance. A tolerance value of -1 indicates the tolerance value is undefined.

Throws

ccShapesError::BadGeom

If **tolType()** != *eAbsolute*, the minimum dimension is defined, and **nomDim()** is not defined. (Measurement conversion is impossible).

maxOffset

```
double maxOffset() const;
```

Returns the relative maximum dimension tolerance. A tolerance value of -1 indicates the tolerance value is undefined.

Throws

ccShapesError::BadGeom

If **tolType()** != *eAbsolute*, the maximum dimension is defined, and **nomDim()** is not defined. (Measurement conversion is impossible).

minPerc

```
double minPerc() const;
```

Returns the minimum dimension tolerance percentage. A tolerance value of -1 indicates the tolerance value is undefined.

Throws

ccShapesError::BadGeom

If **tolType()** != *eAbsolute*, the minimum dimension is defined, and **nomDim()** is not defined. (Measurement conversion is impossible).

maxPerc

```
double maxPerc() const;
```

Returns the maximum dimension tolerance percentage. A tolerance value of -1 indicates the tolerance value is undefined.

Throws

ccShapesError::BadGeom

If **tolType()** != *eAbsolute*, the maximum dimension is defined, and **nomDim()** is not defined. (Measurement conversion is impossible).

limits

```
void limits(double minDim, double maxDim);
void limits(double nomDim, double minDim, double maxDim);
```

- `void limits(double minDim, double maxDim);`

Sets the absolute minimum and maximum dimension tolerances, and sets the tolerance type to *eAbsolute*. A tolerance value of -1 indicates the tolerance value is undefined. The nominal dimension is unchanged.

Parameters

minDim The absolute minimum dimension tolerance.

maxDim The absolute maximum dimension tolerance.

Throws

ccShapesError::BadGeom
If *minDim* < 0 and not -1,
or if *maxDim* < 0 and not -1,
or if *minDim* and *maxDim* are defined and *minDim* > **maxTol()**,
or if *minDim* and *nomDim* are defined and *minDim* > *nomDim*,
or if *maxDim* and *nomDim* are defined and *maxDim* < *nomDim*.

- `void limits(double nomDim, double minDim, double maxDim);`

Sets the nominal dimension, and the absolute minimum and maximum dimension tolerances. Sets the tolerance type to *eAbsolute*. A tolerance value of -1 indicates the tolerance value is undefined.

Parameters

nomDim The nominal dimension value.

minDim The absolute minimum dimension tolerance.

maxDim The absolute maximum dimension tolerance.

Throws

ccShapesError::BadGeom
If *minDim* < 0 and not -1,
or if *maxDim* < 0 and not -1,
or if *minDim* and *maxDim* are defined and *minDim* > **maxTol()**,
or if *minDim* and *nomDim* are defined and *minDim* > *nomDim*,
or if *maxDim* and *nomDim* are defined and *maxDim* < *nomDim*.

offsets

```
void offsets(double minOffset, double maxOffset);

void offsets(
  double nomDim,
  double minOffset,
  double maxOffset);
```

- `void offsets(double minOffset, double maxOffset);`

  Sets the relative minimum and maximum dimension offset tolerances, and sets the tolerance type to *eRelative*. The nominal dimension is unchanged. A tolerance value of -1 indicates the tolerance value is undefined.

  Parameters

  minOffset The relative minimum dimension offset tolerance.

  maxOffset The relative minimum dimension offset tolerance.

  Throws

  ccShapesError::BadGeom
  If *minOffset* < 0 and not -1,
  or if *maxOffset* < 0 and not -1,
  or if *minOffset* and *nomDim* are defined and *minOffset* > *nomDim*.

- ```
  void offsets(
    double nomDim,
    double minOffset,
    double maxOffset);
  ```

  Sets the nominal dimension and the relative minimum and maximum dimension tolerances. Sets the tolerance type to *eRelative*. A tolerance value of -1 indicates the tolerance value is undefined.

  Parameters

  nomDim The nominal dimension.

  minOffset The relative minimum dimension offset tolerance.

  maxOffset The relative minimum dimension offset tolerance.

  Throws

  ccShapesError::BadGeom
  If *minOffset* < 0 and not -1,
  or if *maxOffset* < 0 and not -1,
  or if *minOffset* and *nomDim* are defined and *minOffset* > *nomDim*.
  ```

percentages

```
void percentages(double minPerc, double maxPerc);

void percentages(
  double nomDim,
  double minPerc,
  double maxPerc);
```

Sets the minimum and maximum dimension tolerance percentages, and sets the tolerance type to *ePercentage*. The nominal dimension is unchanged. A tolerance value of -1 indicates the tolerance value is undefined.

- `void percentages(double minPerc, double maxPerc);`

Parameters

minPerc The minimum dimension tolerance percentage.

maxPerc The maximum dimension tolerance percentage.

Throws

ccShapesError::BadGeom
If *minPerc* < 0 and not -1,
or if *minPerc* > 1 and not -1,
or if *maxPerc* < 0 and not -1.

- ```
  void percentages(
    double nomDim,
    double minPerc,
    double maxPerc);
  ```

Sets the nominal dimension and the minimum and maximum dimension tolerance percentages. Sets the tolerance type to *ePercentage*. A tolerance value of -1 indicates the tolerance value is undefined.

Parameters

nomDim The nominal dimension.

minPerc The minimum dimension tolerance percentage.

maxPerc The maximum dimension tolerance percentage.

Throws

ccShapesError::BadGeom
If *minPerc* < 0 and not -1,
or if *minPerc* > 1 and not -1,
or if *maxPerc* < 0 and not -1.
  ```

isWithin

```
bool isWithin(double measurement) const;
```

Returns true if *measurement* is within the absolute minimum and absolute maximum dimension tolerances. More precisely, the function returns true if the following two conditions are both true:

1. **minDim()** == -1 OR *measurement* >= **minDim()**
2. **maxDim()** == -1 OR *measurement* <= **maxDim()**

Parameters

measurement The dimension to be tested.

Throws

ccShapesError::BadGeom
Both nominal dimensions are undefined and **tolType()** != *eAbsolute*.

ccDiscretePelRootPool

```
#include <ch_cvl/discpool.h>

class ccDiscretePelRootPool : public ccPelRootPool;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This class manages an image root pool with a fixed number of same-sized root images. See **ccPelRoot** on page 2391 to learn more about root images.

Constructors/Destructors

ccDiscretePelRootPool

```
ccDiscretePelRootPool(c_Int32 sizeOfEachPelArray,
  void** pelArrays, c_Int32 numArrays = 1,
  c_Int32 minRowUpdate = 0, c_Int32 alignModulus = 32);

ccDiscretePelRootPool(c_Int32 sizeOfEachPelArray,
  const cmStd vector<void*>& pelArrays,
  c_Int32 minRowUpdate = 0, c_Int32 alignModulus = 32);

~ccDiscretePelRootPool();
```

- ```
  ccDiscretePelRootPool(c_Int32 sizeOfEachPelArray,
    void** pelArrays, c_Int32 numArrays = 1,
    c_Int32 minRowUpdate = 0, c_Int32 alignModulus = 32);
  ```

  Constructs a pool of pixel arrays for constructing pel root objects.

  Parameters

  sizeOfEachPelArray
  The size of each root image in the pool.

  pelArrays An array of pointers to the memory allocated for each root image pool. Each *perlArrays[i]* points to a block of *sizeOfEachPelArray* bytes.

  numArrays The number of root images.
  ```

minRowUpdate The minimum row update value to use in bytes. See **cc_PelRoot::rowUpdate()** on page 3473.

alignModulus The alignment modulus value to use in bytes. See **cc_PelRoot::alignModulus()** on page 3474**.**

Notes

The actual row update value used for root images will be no less than *minRowUpdate*, and, if necessary, will be rounded up to be multiple of *alignModulus*.

- ```
  ccDiscretePelRootPool(c_Int32 sizeOfEachPelArray,
    const cmStd vector<void*>& pelArrays,
    c_Int32 minRowUpdate = 0, c_Int32 alignModulus = 32);
  ```

Constructs a pool of pixel arrays for constructing pel root objects. This constructor uses vectors instead of an array of pixel arrays.

Parameters

sizeOfEachPelArray
The size of each root image in the pool.

pelArrays A vector of pointers to the memory allocated for each root image pool. Each *perlArrays[i]* points to a block of *sizeOfEachPelArray* bytes.

minRowUpdate The minimum row update value to use in bytes. See **cc_PelRoot::rowUpdate()** on page 3473.

alignModulus The alignment modulus value to use in bytes. See **cc_PelRoot::alignModulus()** on page 3474**.**

Notes

The actual row update value used for root images will be no less than *minRowUpdate*, and, if necessary, will be rounded up to be multiple of *alignModulus*.

- ```
  ~ccDiscretePelRootPool();
  ```

Destroys the pixel root pool.

Public Member Functions

size

```
virtual void size();

virtual size_t size(size_t bytes) const;
```

- `virtual void size();`

  Returns the number of bytes occupied by the root image pool. This value is equal to the size of each root image multiplied by the number of root images.

- `virtual size_t size(size_t bytes) const;`

  Throws

  ccPelRootPool::NotSupported
  This root image pool does not support resizing.

Typedefs

ccPelRootPoolPtrh_const

```
typedef ccPtrHandle_const<ccPelRootPool>
ccPelRootPoolPtrh_const;
```

A **const** pointer handle to a **ccPelRootPool** object.

ccDiscretePelRootPoolPtrh

```
typedef ccPtrHandle<ccDiscretePelRootPool,
  ccPelRootPoolPtrh, ccPelRootPoolPtrh_const>
  ccDiscretePelRootPoolPtrh;
```

A pointer handle to this class.

ccDiscretePelRootPoolPtrh_const

```
typedef ccPtrHandle_const<ccDiscretePelRootPool,
  ccPelRootPoolPtrh_const>
  ccDiscretePelRootPoolPtrh_const;
```

A **const** pointer handle to this class.

ccDiscretePelRootPool

ccDisplay

```
#include <ch_cvl/display.h>

class ccDisplay : public ccUIControlFrame, public ccRepBase;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is one of the platform-independent interfaces for displaying images and graphics. **ccDisplay** is one of the base classes for the overall display class hierarchy. It serves as the base class for other derived classes that create platform-independent display interfaces, such as **ccWin32Display**. See the following derivation hierarchy:

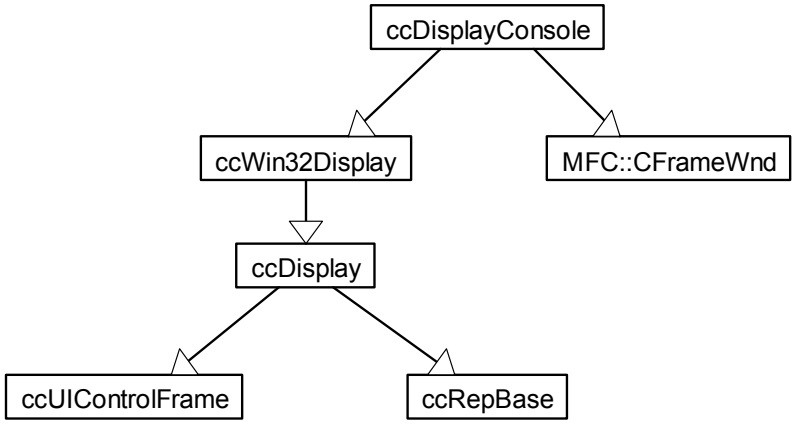


Figure 1. ccDisplay class inheritance hierarchy

Creating your own platform-independent interfaces through direct derivation from **ccDisplay** is not supported. When designing your own display classes, make them subclasses of one of the existing CVL classes that are derived from **ccDisplay**, such as **ccWin32Display** or **ccDisplayConsole**, and overload the existing virtual methods that you class will inherit by this means.

For the majority of CVL applications, including all those that use a Cognex frame grabber, use either the **ccDisplayConsole** or **ccWin32Display** class, as follows:

- **ccWin32Display**: For display classes based on the Microsoft Win32 API for execution on a Windows host computer.
- **ccDisplayConsole**: For applications that use Microsoft MFC and execute on a Windows host computer. **ccDisplayConsole** also inherits from the MFC class **CFrameWnd**.

Note CVL display is not multi-thread safe. Always mutex or semaphore protect any code that accesses a **ccDisplay**-based object and its methods from multiple threads.

Display Features

The **ccDisplay** class contains interfaces that provide the following capabilities:

- Display of different types of pel buffers, such as grey-scale and color. Not all derived implementations support the display of all types of pel buffers. Only one image can be displayed at a time.
- Adding static graphics to the display. Static graphics are added through the **ccUISketch** interface. Once static graphics have been added, the display automatically handles redrawing and refreshing them whenever the system is modified.
- Adding interactive graphics to the display. Interactive graphics are those classes that derive from **ccUIShapes**. Once interactive graphics have been added, the display automatically handles redrawing and refreshing them whenever the system is modified.
- Managing three separate coordinate spaces of the display: client, image, and display coordinates.
- Managing the viewport into the display by providing methods to pan, fit, and scale the image and graphics.
- Scaling the viewport with integer or floating-point precision. The **ccUITablet** interface provides the ability to interpolate images.
- Adding pixel and subpixel grids.
- Application of a user-defined color map that can be used to map image pixel values to color values.
- Control of the shape of the cursor.
- Control of the blinking of graphics colors. Not all derived implementations support the blinking of graphics colors.
- Selection of a region of interest.

- Control of the live display of an acquisition FIFO within the display window.
- Control of synchronization with the monitor refresh rate. Not all derived implementations support this feature.
- Retrieval of a copy of the image and all graphics as they are seen on the screen.

Graphics

Two different types of graphics can be displayed in a **ccDisplay**-derived window:

- **Interactive graphics:** these are **ccUIShapes**-derived objects that are passed into the **addShape()** method, specifying the coordinate system in which they are to be displayed. The *ccUIObject::drawLayer* argument specifies the layer to which the graphic is to be added. These graphics contain handles for manipulating size, rotation, and position.
- **Static graphics:** these are non-interactive graphics. They are created in a **ccUISketch** and passed into the **drawSketch()** method, specifying the coordinate system in which they are to be displayed. A sketch is a list of drawing commands that are recorded into a **ccUITablet** when recording is turned on. See the **ccUITablet** reference page for more information.

The static representation of a graphic generally renders faster and uses fewer resources than its interactive counterpart (for example, **ccCircle** is faster than **ccUICircle**).

Coordinate Systems

Three different coordinate systems are available within CVL display:

- **Display coordinates:** used to draw graphics in the display coordinate system. The display coordinate system is relative to the upper left corner of the display window.
- **Image coordinates:** used to draw graphics in the image coordinate system. The image coordinate system is relative to the upper left corner of the image. Mapping from image to display coordinates is based on translation and scale, and depends on whether the image is panned or zoomed, and whether it has an offset. When there is no image in the display, the image coordinate space is the same as the display coordinate space.
- **Client coordinates:** used to draw graphics in the client coordinate system. The transform contained within the pel buffer that is passed to the **image()** method specifies how image coordinates are to be translated to client coordinates.

Constructors/Destructors

The constructors and destructors of **ccDisplay** are used exclusively by derived classes. Do not instantiate **ccDisplay** objects directly in your own applications.

Enumerations

DisplayFormat `enum DisplayFormat`

This enumeration specifies the display format of the display. For **ccWin32Display** and **ccDisplayConsole**, the **displayFormat()** method returns the desktop format. For all other classes derived from **ccDisplay**, **displayFormat()** returns *e8Bit*.

Value	Meaning
e8Bit	The image is an 8-bit grey scale image (8 bits per pixel).
eRGB15	The image is a 15-bit RGB image (15 bits per pixel, encoded as `0rrr rrgg gggb bbbb`). This format is also known as packed 5-5-5 RGB.
eRGB16	The image is a 16-bit RGB image (16 bits per pixel, encoded as `rrrr rggg gggb bbbb`). This format is also known as packed 5-6-5 RGB.
eRGB24	The image is a 24-bit RGB image (24 bits per pixel, the bytes encoded as `0rgb`). This format is also known as 8-8-8 RGB.
eRGB32	The image is a 32-bit RGB image (32 bits per pixel, the bytes encoded as `argb`, where a is an alphabetic byte). This format is also known as packed 0-8-8-8 ARGB.

CoordinateSystem

```
enum CoordinateSystem
```

This enumeration lets you specify the coordinate system to use for display operations.

Value	Meaning
eDisplayCoords	Display coordinate space. This coordinate system is relative to the upper left corner of the display window, irrespective of panning or zooming.
eImageCoords	Image coordinate space. This coordinate system is relative to the upper left corner of the image when the transform of the pel buffer is the identity transform. The mapping from image to display coordinates can include translation and scale information, depending on whether the image is panned or zoomed, or both, and whether it has an offset. When there is no image in the display, the image coordinate space is the same as the display coordinate space.
eClientCoords	Client coordinate space. The image-to-client coordinate system transformation is specified by the transform object associated with the pel buffer. When this is the identity transform, the client coordinate space is the same as the image coordinate space.

ColorMapIndex

```
enum ColorMapIndex
```

This enumeration defines the minimum and maximum indices into the entire color map over which you have control. You cannot change the color map setting for indices less than *eMinUserColor* or greater than *eMaxUserColor*. Note that the default value of 0 for the *mapStart* argument to the **colorMap()** setter causes the input map to be used starting at index *eMinUserColor*, or [10].

Value	Meaning
eMinUserColor = 10	Minimum index into the area of the color map that is user-accessible (10).
eMaxUserColor =240	Maximum index into the area of the color map that is user-accessible (240).

MouseMode `enum MouseMode`

This enumeration specifies the mouse mode.

Value		Meaning
ePointer		Sets the cursor to a pointer arrow shape. Graphics can be selected, panned, and so on. The image cannot be panned. Switches to *ePan* mode when the *<Ctrl>* key or the *<Shift>* key is pressed.
ePan		Sets the cursor to a hand shape. Graphics cannot be selected or panned. The image can be panned. Switches to *ePointer* mode when the *<Ctrl>* key or the *<Shift>* key is pressed.
eZoomIn		Sets the cursor to a plus magnifying glass shape. Graphics cannot be selected or panned. The image cannot be panned. Left mouse click zooms in, right mouse click zooms out. Switches to *ePan* mode when the *<Ctrl>* key or the *<Shift>* key is pressed.
eZoomOut		Sets the cursor to a minus magnifying glass shape. Graphics cannot be selected or panned. The image cannot be panned. Left mouse click zooms out, right mouse click zooms in. Switches to *ePan* mode when the *<Ctrl>* key or the *<Shift>* key is pressed.

overlayState `enum overlayState;`

This enumeration enables or disable the overlay layer of the drawing tablet (*ccUITablet::eOverlayLayer*).

Value	Meaning
eOverlayPlane	Enables the overlay layer. This is a complementary value to *eDisableOverlayPlane*.
eChromaKeying	Chromakeying is deprecated. Do not use this value in new code.

Value	Meaning
eDisableOverlayPlane	Disables the overlay layer. This is a complementary value to *eOverlayPlane*. When the overlay layer is disabled, any sketches and interactive shapes added to the overlay layer are discarded. This value applies only to host-based displays (**ccDisplayConsole** and **ccWin32Display**).
eDisableChromaKeying	Chromakeying is deprecated. Do not use this value in new code.
eNone	The overlay layer is not supported. Never pass this value as an argument to the **enableOverlay()** setter. The **enableOverlay()** getter returns this value when both the overlay layer and chromakeying are disabled.

Notes

overlayState values cannot be OR'ed together.

Public Member Functions

image

```
void image(const ccPelBuffer_const<c_UInt8>& pb,
  bool displayRaw = true);

void image(const ccPelBuffer_const<ccPackedRGB16Pel>& pb);

void image(const ccPelBuffer_const<ccPackedRGB32Pel>& pb);

bool image(const ccAcqImagePtrh, cc2Xform* img2client=0);
```

Displays the image in the supplied pel buffer.

On 8-bit desktops, pixel values in the range [10,240] are displayed as grey values and pixel values in the ranges [0,9] and [241,255] are displayed as colors by default. See also **colorMap()**.

On non 8-bit desktops, all pixel values are displayed as grey values by default. See also **colorMapEx()**.

This method uses the image from client coordinate transform, produced by **imageFromClientXform()** and stored in the pel buffer, to convert client coordinates to image coordinates. Call **image()** again to redisplay the image if the transform changes.

The base map of the drawing tablet, retrieved with **ccUITablet::baseMap()**, is automatically updated to reflect the current state of the image-from-client-coordinate transform of the image. When the transform is the identify matrix, the image is automatically centered within the display.

Whenever the image size (width or height) or transform differs from that of the current displayed image, all graphics are automatically redrawn on all layers.

Each time you call **image()**, the current image in the display is replaced with the supplied image, and all static and interactive graphics on the image layer are re-rendered. Graphics on the overlay layer are not redrawn, unless the transform has changed.

If the supplied pel buffer contains a non-linear transform, the display linearizes the transform using a first order polynomial fit over the entire area of the image. Any client coordinate graphics already contained in the display then use this linear approximation.

Notes

This method triggers an **imageChanged()** event to notify that a new image is about to be displayed. It also triggers an **updatePanRanges()** event when **defaultPan()** is true. See *Protected Member Functions* of **ccDisplay** for more information.

- `void image(const ccPelBuffer_const<c_UInt8>& pb, bool displayRaw = true);`

  Displays the supplied 8-bit pel buffer. This overload provides an option to clip pixel values.

  Parameters

  pb The 8-bit pel buffer.

  displayRaw If true, the pel buffer is not clipped. All pixel values are displayed as is with no mapping of pixel values. This is the default behavior.

  If false, the pel buffer is clipped. Values are mapped as follows:
  Pixel values in the range [0,9] are mapped to 10.
  Pixel values in the range [241,255] are mapped to 240.
  All other pixel values remain unchanged.

  Notes

  The *displayRaw* parameter is applicable to all desktop depths.

  An 8-bit pel buffer can be displayed on any desktop depth. If the physical display device has a non-8-bit pixel format, the pel buffer is converted automatically, with a corresponding increase in execution time.

- `void image(const ccPelBuffer_const<ccPackedRGB16Pel>& pb);`

  Displays the supplied 16-bit RGB pel buffer. Uses the transform stored in *pb*. This method must be called again if the transform changes.

  Parameters

  pb The 16-bit RGB pel buffer.

  Notes

  The image must be in packed 5-6-5 RGB format (that is, **displayFormat()** must return *eRGB16* for this image). The desktop depth of the physical display device must also be set to 16-bit to use this overload.

  The color map has no effect on the image displayed by this function.

- `void image(const ccPelBuffer_const<ccPackedRGB32Pel>& pb);`

  Displays the supplied 32-bit RGB pel buffer. Uses the transform stored in *pb*. This method must be called again if the transform changes.

  Parameters

  pb The 32-bit RGB pel buffer.

Notes

The image format must be in 32-bit RGB (that is, **displayFormat()** must return *eRGB32* for this image). The desktop depth of the physical display device must also be set to 32-bit to use this overload.

The color map has no effect on the image displayed by this function.

- `bool image(const ccAcqImagePtrh, cc2Xform* img2client=0);`

Displays an acquired image. The image is first converted to the proper pel buffer type for the current display settings. The transform in img2client, if specified, is then applied to the pel buffer. Finally, the pel buffer is displayed using the other other overloads of the image() function, as described above.

Returns true if the image was successfully displayed. Returns false if the image cannot be converted to a format suitable for display.

Notes

Color images will be displayed in color, if possible.

displayFormat `virtual DisplayFormat displayFormat() const;`

Retrieves the display format of this display.

hasImage `virtual bool hasImage() const;`

Returns true if the display has an image and it is currently visible.

hasImage8 `virtual bool hasImage8() const;`

Returns true if the display has an 8-bit image that it will manage. The image need not be currently visible.

Notes

Only one of the **hasImage*()** methods can be true at a time, as the CVL display framework can display and manage only a single image at a time.

hasImage16 `virtual bool hasImage16() const;`

Returns true if the display has a 16-bit image that it will manage. The image need not be currently visible.

Notes

Only one of the **hasImage*()** methods can be true at a time, as the CVL display framework can display and manage only a single image at a time.

hasImage32 `virtual bool hasImage32() const;`

Returns true if the display has a 32-bit image that it will manage. The image need not be currently visible.

Notes

Only one of the **hasImage*()** methods can be true at a time, as the CVL display framework can display and manage only a single image at a time.

removeImage `void removeImage();`

Removes the image from the display and resets the display to show the background only. All graphics in the display are automatically redrawn.

Notes

All of the **hasImage*()** methods return false after this method has been called.

imageSize `virtual ccIPair imageSize() const;`

This method returns the size of the image as a (width, height) pair if **hasImage()** is true. If **hasImage()** is false, this method returns (0,0).

Notes

The image size may be larger, smaller, or equal to the client area of the display.

imageOffset `virtual ccIPair imageOffset() const;`

This method returns the offset of the image if **hasImage()** is true. If **hasImage()** is false, this method returns (0,0).

clientArea `virtual ccIPair clientArea() const;`

Returns the current size of the client area of the display.

Notes

The size is returned as an integer pair (width, height). This size represents only the client drawing area and does not include any window decorations such as borders and scroll bars. This size may not necessarily match the size of the image currently in the display.

addShape

```
void addShape(ccUIShapes* shape, CoordinateSystem cs);
```

Adds the specified shape to the display. The shape is drawn in the specified coordinate system. Use this method with interactive shapes.

The drawing layer is specified in the shape itself (default = the image layer). You can change the layer by calling the shape's **ccUIObject::drawLayer()** method before calling this method.

Parameters

shape The interactive shape to add to the display.

cs The coordinate system to use to draw the shape. Must be one of the following values:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

See *CoordinateSystem* on page 1193 for more information.

Notes

In CVL, there are two ways to display graphics:

- Create UI shapes derived from **ccUIShapes** and pass them to **addShape()**, specifying the coordinate system in which to display the shape.
- Create a sketch and pass it to **drawSketch()**, again specifying the coordinate system. A sketch is a list of recorded drawing commands, recorded by drawing into a **ccUITablet** with recording turned on. You can either call the tablet's draw methods directly, or call the **ccUIShape**-derived objects' **draw()** methods by passing in a reference to the tablet.

Use **removeShape()** to remove interactive shapes from the display. You can add a shape to a different display without calling **removeShape()**. You can add a UI shape to only one display at a time.

If the UI shape's *ccUIObject::autoDelete* flag is set to true, then this display object takes ownership of the shape when it is added. Upon termination, the display automatically deletes any shapes it owns that were not previously removed using **removeShape()**. Therefore, UI shapes for which the *ccUIObject::autoDelete* flag is true must be allocated on the heap (with **new**) and must not be destroyed (with **delete**) in user code. You must explicitly delete only those UI shapes for which the *ccUIObject::autoDelete* flag is set to false.

The display automatically redraws interactive shapes as needed, re-rendering all other shapes in the display each time a new shape is added. Therefore, when adding a large number of shapes, call **disableDrawing()** before adding the first

shape, and call **enableDrawing()** after all the shapes have been added. This will improve performance, as the shapes will not be rendered until **enableDrawing()** has been called.

See **drawSketch()** for information on adding static graphics.

removeShape

```
bool removeShape(ccUIShapes* shape);
```

Removes the specified shape from the display. Returns true if the operation is successful, and false if the shape does not belong to this display. Use this method with interactive shapes.

Parameters

shape The interactive shape to remove from the display. This parameter must not be a NULL pointer.

Notes

You may have to call either **root()->globalUpdate()** or **ccUITablet::fullDraw()** after calling this function to redraw any overlapping shapes. If you remove a shape that overlaps other shapes, those other shapes may be partially erased. Calling one of these methods after each invocation of **removeShape()** will ensure that the display is correct. However, be aware that this will add overhead to your application.

When removing a large number of shapes, call **disableDrawing()** before calling **removeShape()**. Call **enableDrawing()** after all the shapes have been removed. This will improve performance, as the shapes will not be erased until **enableDrawing()** is called.

See **eraseSketch()** for information on removing static graphics.

drawSketch

```
void drawSketch(const ccUISketch& sketch,
 CoordinateSystem cs);

void drawSketch(const ccUISketch& sketch,
 CoordinateSystem cs, const ccRect& clipRect);
```

Draws the sketch in the specified coordinate system, and appends it to any other sketches that have already been added to the display for that coordinate space. A cliprect can be specified to restrict the rendering of the given sketch to within the rectangular clipping region.

Use this method with static graphics.

Parameters

sketch The sketch to draw.

cs — The coordinate system to use to draw the sketch. Must be one of the following values:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

See *CoordinateSystem* on page 1193 for more information.

clipRect — Specifies a restricted bounding box into which the given sketch will be rendered. The units of the *clipRect* should be in the same units as the graphics and whose reference point will be based on the coordinate system specified. The clipping region is always axis-aligned, and as such, the *clipRect* can be represented as an axis-aligned bounding box. fi the *clipRect* is the null rect (**ccRect**(0,0,0,0)), then no clipping is performed.

The *clipRect* must not degenerate. See **ccRect::degen()**.

Notes

The *clipRect* only applies to the specified *sketch* and not any other sketches that may already be added to the display system.

The *clipRect* applies to all graphics contained within the given *sketch*. The *clipRect* does not depend on the layer the graphic is associated with. The *clipRect* is applied to both the image layer and the overlay layer when viewing the display window.

By default, the display system only renders sketch graphics that are contained within the visible viewing area of the display window. By specifying a *clipRect*, you can alter this default behavior and restrict the rendering of sketch graphics to the

bounding box specified by *clipRect*. In cases where the *clipRect* intersects the display viewing area, the clipping region consists of the intersection of the *clipRect* and the display viewable area.

The graphics contained within the sketch are drawn using the current zoom and scroll settings, and will be automatically redrawn as needed whenever the zoom, scroll, or other part of the display mechanism requires a redraw.

All graphics in the sketch specified in the **ccUITablet::eImageLayer** and **ccUITablet::eOverlayLayer** are immediately rendered to the display. On some platforms, you must enable the overlay layer in order to see graphics on that layer, otherwise graphics on the **ccUITablet::eOverlayer** may not be visible. See **ccDisplay::enableOverlay**.

If drawingDisabled() == false, then the sketch is immediately rendered into the display.

Not all platforms support clipping of sketches. For those platforms that do not support clipping, the clipRect argument is ignored and rendering proceeds as if no clipping rectangle was present.

See **addShape()** for information on adding interactive graphics.

eraseSketch

```
void eraseSketch(CoordinateSystem cs);
```

Erases the sketch currently displayed in the specified coordinate system. Use this method with static graphics.

Parameters

cs The coordinate system to use when erasing the sketch. Must be one of the following:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

See *CoordinateSystem* on page 1193 for more information.

Notes

Calling this method may cause a complete refresh of other parts of the display, including redrawing any interactive graphics in the display.

See **removeShape()** for information on removing interactive graphics.

getSketch

```
ccUISketch& getSketch(CoordinateSystem cs);
```

Returns a reference to the sketch in the specified coordinate system.

Parameters

cs The coordinate system to use when getting the sketch. Must be one of the following values:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

See *CoordinateSystem* on page 1193 for more information.

Notes

Use extreme caution when calling this method. The return value is a non-**const** reference, which will allow you to change the internal sketch separately from the display. Changing a sketch while the display is using it may cause unexpected behavior.

If the returned sketch is passed into the **drawSketch()** method that takes a clipping rectangle, then the clipping of graphics to the clipping region may not apply if the sketch already contains a clipping region.

Be sure to mutex/semaphore protect any code using **getSketch()**, or any other CVL display methods, when using display from multiple threads.

disableDrawing `void disableDrawing();`

Disables drawing and prevents screen updates, including all static and interactive graphics on all coordinate systems. Call **disableDrawing()** and then **enableDrawing()** to refresh the entire display.

For every call to **disableDrawing()**, there must be a corresponding call to **enableDrawing()** to re-enable drawing. These two methods must be called in pairs, with **disableDrawing()** always being called before **enableDrawing()**. Nesting of these paired methods is allowed, but the nesting cannot exceed 32 levels.

Throws

ccUIError::FlagStackOverflow
disableDrawing() was called more than 32 times without **enableDrawing()** having being called.

Notes

Call **disableDrawing()** to prevent multiple redraws when making multiple changes to the display, such as changing the image and then inserting a sketch, or changing the pan and then the magnification setting.

enableDrawing `void enableDrawing (bool redraw = true);`

Cancels the effect of one call to **disableDrawing()**. See **disableDrawing()** above.

Parameters

redraw If true, the display is redrawn by calling **ccUITablet::fullDraw()**.

Throws

ccUIError::FlagStackUnderflow
No matching call to **disableDrawing()** was made.

drawingDisabled

```
bool drawingDisabled() const;
```

Returns true if drawing is disabled, false otherwise.

Notes

The **disableDrawing()** and **enableDrawing()** methods control the drawing state.

---

pan

```
void pan(const cc2Vect& offset);

const cc2Vect& pan() const;
```

- `void pan(const cc2Vect& offset);`

  Sets the pan value. Positive values for **panOffset.x()** and **panOffset.y()** shift the image down and to the right. The pan offset is the distance from the center of the display to the center of the image in image coordinates.

  Parameters

  offset The amount by which to pan in image coordinates.

  Notes

  Setting *offset* to (0,0) centers the image in the display.

  This setter triggers a **panChanged()** event to notify that a new pan setting has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

- `const cc2Vect& pan() const;`

  Returns the current pan setting.

---

maxPan

```
ccDPair maxPan() const;

void maxPan(const ccDPair& xy);
```

- `ccDPair maxPan() const;`

  Returns the maximum pan value.

- `void maxPan(const ccDPair& xy);`

  Sets the maximum pan value.

  Parameters

  xy The maximum pan value.

  Notes

  This setter triggers an **updatePanRanges()** event to notify that a new maximum pan setting has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

minPan

```
ccDPair minPan() const;
void minPan(const ccDPair&);
```

- `ccDPair minPan() const;`

  Returns the minimum pan value.

- `void minPan(const ccDPair& xy);`

  Sets the minimum pan value.

  Parameters

  xy The minimum pan value.

  Notes

  This setter triggers an **updatePanRanges()** event to notify that a new minimum pan setting has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

defaultPan

```
bool defaultPan() const;
void defaultPan(bool defaultToImageSize);
```

- `bool defaultPan() const;`

  Returns true if panning is limited to the image size.

- `void defaultPan(bool defaultToImageSize);`

  Sets panning behavior.

Parameters

defaultToImageSize

If true, the minimum and maximum pan values are limited to the image size. If true, the pan ranges are automatically updated whenever a new image is added to the display. If false, the pan ranges are not automatically updated.

Notes

To adjust the minimum and maximum pan ranges, use **minPan()** and **maxPan()**, respectively.

panDelta

```
void panDelta(const ccIPair& delta);
```

Pans the image by the specified amount in display coordinates.

Parameters

delta The amount by which to pan the image in the x and y directions in display coordinates.

mag

```
void mag(c_Int32 m);

c_Int32 mag() const;
```

- `void mag(c_Int32 m);`

Sets the magnification (zoom) value. The display is completely refreshed, and all graphics are re-rendered at the new magnification level.

Parameters

m An integer-based magnification value. Positive values zoom in, negative values zoom out. Thus, a value of 2 means 2x magnification and a value of -2 means 1/2x magnification.

Notes

This setter triggers a **magChanged()** event to notify that a new magnification setting has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

- `c_Int32 mag() const;`

Retrieves the current magnification (zoom) value.

magExact

```
void magExact(double m);
double magExact() const;
```

- `void magExact(double m);`

  Sets the magnification (zoom). This method allows you to specify a **double** as the scale value. The display is completely refreshed, and all graphics are re-rendered at the new magnification level.

  Parameters

  m — A floating-point magnification value. A value of 1.52 means 152% of the actual size. A value of 0.5 means 50% of the actual size. This value must be non-negative.

  Notes

  The image is scaled as close to the specified value as possible. Non-integer scale values force sampling of the image, which can significantly decrease performance when panning and in live display. To restore performance, use the **mag()** method.

  This setter triggers a **magChanged()** event to notify that a new magnification setting has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

- `double magExact() const;`

  Gets the current magnification scale.

fit

```
void fit();
```

Centers the image and changes the magnification to the integer scale value that best fits the current display window size.

fitExact

```
void fitExact();
```

Centers the image in the display and changes the magnification to best fit the current display window size.

Notes

One dimension of the image, either x or y, will fit exactly.The other will fit as close as possible so that the aspect ratio is preserved and the entire image is viewable.

grid

```
void grid(c_Int32 spacing);
c_Int32 grid() const;
```

- `void grid(c_Int32 spacing);`

  Sets the grid spacing, in pixels.

  Parameters

  spacing — The number of pixels per grid lines. A value of 1 places a grid line at every pixel boundary. A value of 2 places a grid line at every other pixel, and so on. Must be a positive value for grid lines to be displayed.

  Notes

  No grid lines are drawn if spacing is less than or equal to 0, or if the image is not sufficiently magnified.

- `c_Int32 grid() const;`

  Returns the current grid spacing, in pixels.

gridColor

```
void gridColor(const ccColor& c);
const ccColor& gridColor() const;
```

- `void gridColor(const ccColor& c);`

  Sets the color of the pixel grid lines.

  Parameters

  c — The color of the grid lines.

- `const ccColor& gridColor() const;`

  Returns the current color of the pixel grid lines.

subpixelGrid

```
void subpixelGrid(c_Int32 spacing);
c_Int32 subpixelGrid() const;
```

- `void subpixelGrid(c_Int32 spacing);`

  Sets the sub-pixel grid spacing.

Parameters

spacing A spacing of *n* means draw *n-1* grid lines within each pixel. A value less than 2 means no grid lines.

Notes

No grid lines are drawn if the image is not magnified enough.

- `c_Int32 subpixelGrid() const;`

Returns the current sub-pixel grid spacing.

subpixelGridColor

```
void subpixelGridColor(const ccColor& c);

const ccColor& subpixelGridColor() const;
```

- `void subpixelGridColor(const ccColor& c);`

Sets the color of the sub-pixel grid lines.

Parameters

c The color of the grid lines.

- `const ccColor& subpixelGridColor() const;`

Returns the current color of the sub-pixel grid lines.

colorMap

```
cmStd vector<ccRGB> colorMap() const;

void colorMap(const cmStd vector<ccRGB>& map,
  c_Int32 mapStart = 0);
```

- `cmStd vector<ccRGB> colorMap() const;`

Returns the color map. The color map is a 256-entry lookup table that controls image pixel mapping, and graphics color mapping in the reserved ranges, on 8-bit desktops. This method returns the entire lookup table, including both the reserved and user-accessible ranges.

Notes

This method does not return the lookup table used for pixel mapping image pel buffers on non-8-bit desktops. Cognex therefore recommends using this method only for 8-bit desktops.

See the *Color Maps* section of the *Displaying Images* chapter in the *CVL User's Guide* for more information.

- ```
  void colorMap(const cmStd vector<ccRGB>& map,
    c_Int32 mapStart = 0);
  ```

Sets the lookup table used for mapping graphics and images. Maps from 8-bit pseudo color to RGB. The lookup table has 256 entries, of which entries [0,9] and [241,255] are fixed and cannot be changed. You can modify entries [10,240].

Parameters

map — The pseudo color to RGB map, specifying the new RGB values to insert into the table.

mapStart — The location in *map* to be used as the starting point for copying into the 256-entry lookup table starting at index [10].

Notes

This method is supported for all desktop depths. However, Cognex recommends using this method only for 8-bit desktops. To set the lookup table on non-8-bit desktops, use **colorMapEx()** instead.

This setter triggers a **colorMapChanged()** event to notify that a new color map for 8-bit desktops has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

See the *Color Maps* section of the *Displaying Images* chapter in the *CVL User's Guide* for more information on color maps.

colorMapEx

```
cmStd vector<ccRGB> colorMapEx() const;

void colorMapEx(const cmStd vector<ccRGB>& map);
```

- ```
  cmStd vector<ccRGB> colorMapEx() const;
  ```

Returns the image color map used with non-8-bit desktops. This function returns the entire lookup table.

See the *Color Maps* section of the *Displaying Images* chapter in the *CVL User's Guide* for more information.

- ```
  void colorMapEx(const cmStd vector<ccRGB>& map);
  ```

Sets the image color map used with non-8-bit desktops. This function sets the entire 256-element range of the color map. This function is supported only for non-8-bit desktops.
  ```

Parameters

map Vector of RGB colors to map to the look-up table. Must contain 256 elements.

Throws

ccUIError::InvalidColorMapSize
The size of *map* is not 256.

Notes

This method is supported only for non-8-bit desktops. It is not supported for 8-bit desktops.

This setter triggers an **imageMapChanged()** event to notify that a new color map for non-8-bit desktops has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

See also the *Color Maps* section of the *Displaying Images* chapter in the *CVL User's Guide* for more information on color maps.

clearColorMap `void clearColorMap();`

Sets all values in the user-accessible range of the graphics color map (that is, the range [10, 240]) to their default, index-based, grey-scale RGB values:

```
for(i = eMinUserColor; i <= eMaxUserColor; i++)
  colorMap[i] = ccRGB(i, i, i);
```

For example, colorMap[10] is set to the RGB values 10, 10, 10, or very dark grey; and colorMap[240] is set to the RGB values 240, 240, 240, or very light grey.

Notes

This method applies only to the lookup table used for graphics mapping. To set the lookup table used for image mapping, use **colorMap()**.

overlayColorMap

```
cmStd vector<ccRGB> overlayColorMap() const;
```

Returns the overlay color map. The overlay color map is a lookup table that maps between pixel values in an overlay display buffer and **ccRGB** values. (See the **ccRGB** reference page for information about **ccRGB** values).

The size of the returned vector is both platform and hardware-configuration dependent. There is a one-to-one correspondence between the returned vector index and pixel values in the overlay layers display buffer. For example, pixel values of 0 map to the **ccRGB** value found at vector index 0, pixel values of 1 map to the **ccRGB** value found at vector index 1, and so on.

The length of the returned vector is one more than the maximum pixel value that can exist in the overlay display buffer. Use **ccDisplay::getDisplayedImage()** to retrieve the overlay display buffer.

See also the *Color Maps* section of the *Displaying Images* chapter in the *CVL User's Guide* for more information on color maps.

Throws

ccUIError::NotAvailable

This display does not support overlay buffers. Use **ccUITablet::overlaySupported()** to test for overlay buffer support.

ccUIError::NotEnabled

This display supports overlay buffers, but the overlay layer is not currently enabled. Some classes derived from **ccDisplay** require the overlay display buffer to be explicitly enabled.

getPassThroughValue

```
virtual void getPassThroughValue(c_UInt8& v) const;

virtual void getPassThroughValue(ccPackedRGB16Pel& v)
 const;

virtual void getPassThroughValue(ccPackedRGB32Pel& v)
 const;
```

The pass through value is the value used when the overlay display buffer is combined with the image display buffer such that any pixel value in the overlay display buffer that is equal to the pass through value will show through, revealing the corresponding image display buffer pixel underneath. For example, if your overlay display buffer contains no

pass through values, you will display only the overlay buffer and none of the image buffer. If your overlay display buffer contains all pass through values, you will display only the image buffer and none of the overlay buffer.

All overlay display buffers are initialized with a pass through value depending on the current desktop depth.

Notes

Some **ccDisplay** subclasses may require the overlay display buffer to be explicitly enabled. However, the overlay display buffer does not need to be enabled for this function to perform properly.

- `virtual void getPassThroughValue(c_UInt8 & v) const;`

Returns the 8-bit pass through value used with overlay display buffers.

Parameters

v Specifies the pixel depth.

Throws

ccUIError::NotImplemented
The derived class does not support this format.

- `virtual void getPassThroughValue(ccPackedRGB16Pel& v) const;`

Returns the 16-bit pass through value used with overlay display buffers.

Parameters

v Specifies the pixel depth.

Throws

ccUIError::NotImplemented
The derived class does not support this format.

- `virtual void getPassThroughValue(ccPackedRGB32Pel& v) const;`

Returns the 32-bit pass through value used with overlay display buffers.

Parameters

v Specifies the pixel depth.

Throws

ccUIError::NotImplemented
The derived class does not support this format.

mouseMode

```
virtual void mouseMode(MouseMode mm);

MouseMode mouseMode() const;
```

- `virtual void mouseMode(MouseMode mm);`

  Sets the mouse mode. The mouse mode specifies the shape of the mouse cursor and the action that takes place when the user clicks in a display.

  Parameters

  mm The mouse mode. Must be one of the following values:

  ccDisplay::ePointer
  ccDisplay::ePan
  ccDisplay::eZoomIn
  ccDisplay::eZoomOut

  Notes

  This setter triggers a **mouseModeChanged()** event to notify that a new mouse mode has been applied. See *Protected Member Functions* of **ccDisplay** for more information.

  See also *MouseMode* on page 1194 for more information on this enumerated value.

- `MouseMode mouseMode() const;`

  Returns the current mouse mode.

selectAreaStart

```
virtual c_Int32 selectAreaStart(
  CoordinateSystem cs = eImageCoords,
  const ccPelRect& startLocation = ccPelRect(),
  const ccColor& c = ccColor::greenColor()) = 0;
```

Starts a rectangular area selection. Displays a manipulable rectangle that the user can size and reposition with the mouse. Returns a unique tag number indicating the selection. The tag number can be passed to **ccDisplay::selectAreaEnd()**.

Parameters

cs The coordinate system to use. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation The initial location of the selection area in the coordinate system specified by *cs*. If *startLocation* is a null rectangle, degenerate, or does not overlap the visible portion of the image, the initial

location of the manipulable rectangle is either:
- The location of the most recently selected area, or
- Centered within an area that is two-thirds the size of the client area of the display.

c — The color to use to draw the selection rectangle. Can be any of the predefined stock colors (defined in *<ch_cvl/color.h>*) or a user-construced **ccColor**. The default color is green.

Throws

ccUIError::NotImplemented
The derived class does not support this function.

selectAreaEnd

```
virtual ccPelRect selectAreaEnd(c_Int32 tag) = 0;
```

Completes the rectangular area selection started by **ccDisplay::selectAreaStart()**. Hides the rectangle graphic. Returns the selected area in the coordinate system specified in **ccDisplay::selectAreaStart()**.

Parameters

tag — The tag identifying the selection.

Throws

ccUIError::NotImplemented
The derived class does not support this function.

Notes

If *tag* is not a value returned by **selectAreaStart()**, this function has no effect.

selectArea

```
virtual ccPelRect selectArea(
  CoordinateSystem cs = eImageCoords,
  const ccPelRect& startLocation = ccPelRect(),
  const TCHAR* prompt = cmT("Select an Area"),
  const ccColor& c = ccColor::greenColor()) = 0;
```

Starts and ends an area selection. Displays a manipulable rectangle that the user can size and reposition with the mouse. Waits for the user to select a rectangular area on the display and to confirm the selection by pressing the OK button in a message dialog box. When the user presses OK, this method returns the selected rectangle in the specified coordinate system. The user can cancel the selection by closing the message dialog box.

Parameters

cs — The coordinate system to use for reporting the location of the selected rectangle. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation The initial location of the selection area in the coordinate system specified by *cs*.

If *startLocation* is a null rectangle, degenerate, or does not overlap the visible portion of the image, the initial location of the manipulable rectangle is either:
- The location of the most recently selected area, or
- Centered within an area that is two-thirds the size of the client area of the display.

prompt The message displayed in the message dialog box. The default prompt is "Select an Area."

c The color used to draw the selection rectangle. Can be any of the predefined stock colors (defined in *<ch_cvl/color.h>*, or a user-constructed color. The default color is green.

Throws

ccUIError::NotImplemented
The derived class does not support this function.

Notes

This method is usually implemented as a wrapper for **selectAreaStart()** and **selectAreaEnd()**.

selectPointStart

```
virtual c_Int32 selectPointStart(
  CoordinateSystem cs = eImageCoords,
  const ccDPair& startLocation = ccDPair(-1,-1),
  const ccColor& c = ccColor::greenColor()) = 0;
```

Starts a point selection. Displays a **ccUIPointIcon** graphic that the user can reposition with the mouse.

Returns a tag number indicating the selection. The tag number can be passed to **selectPointEnd()**.

Parameters

cs The coordinate system to use. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords

startLocation The initial location of the point icon in the coordinate system specified by *cs*.

If *startLocation* is not in the visible portion of the image, the initial location of the point icon is either:
- The location of the last-selected point, if the coordinate system from the last selected point matches that specified by *cs*, or
- Centered within the client area of the display if the coordinate system of the last-selected point differs from that specified by *cs*.

c — The color to use to draw the selection cross. Can be any of the predefined stock colors (defined in <ch_cvl/*color.h*>) or a user-constructed color. The default color is green.

Throws

ccUIError::NotImplemented
The derived class does not support this function.

selectPointEnd

```
virtual ccDPair selectPointEnd(c_Int32 tag) = 0;
```

Completes a point selection started with **selectPointStart()**. Hides the **ccUIPointIcon** graphic, and returns the center of the point icon representing the selected point in the coordinate system specified in **selectPointStart()**.

Parameters

tag — The tag that identifies the selection. If *tag* is not a value returned by **selectPointStart()**, this function has no effect.

Throws

ccUIError::NotImplemented
The derived class does not support this function.

selectPoint

```
virtual ccDPair selectPoint(
  CoordinateSystem cs = eImageCoords,
  const ccDPair& startLocation = ccDPair(-1,-1),
  const TCHAR* prompt = cmT("Select a Point"),
  const ccColor& c = ccColor::greenColor()) = 0;
```

Starts and ends a point selection. Displays a **ccUIPointIcon** graphic and waits for the user to select a point on the display (see **selectPointStart()** above). This function does not return until the user confirms the selection by pressing the OK button in a message dialog box. The user may cancel the operation instead of selecting.

When the user presses OK, this function returns the point in the center of the point icon in the specified coordinate system.

Parameters

cs — The coordinate system to use. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation The initial location of the selection cross in the coordinate system specified by *cs*. If *startLocation* is not in the visible portion of the image, the initial location is either:
- The location of the last-selected point if the coordinate system of the last-selected point matches that specified by *cs*, or
- Centered within the client area of the display if the coordinate system of the last-selected point differs from that specified by *cs*.

prompt The title displayed in the message dialog box.

c The color used to draw the point icon. Can be any of the predefined stock colors (defined in *<ch_cvl/color.h>*), or a user-constructed color. The default color is green.

Throws

ccUIError::NotImplemented
The derived class does not support this function.

Notes

This method is usually implemented as a simple wrapper for **selectPointStart()** and **selectPointEnd()**.

image8

```
const ccPelBuffer_const<c_UInt8>& image8();
```

Returns a reference to the 8-bit grey-scale image currently in the display.

Notes

The display can manage only one image at a time. Therefore, only one image can be bound to this display at a given time.

imageRGB16

```
const ccPelBuffer_const<ccPackedRGB16Pel>& imageRGB16()
  const;
```

Returns a reference to the 16-bit color image currently in the display.

Notes

The display can manage only one image at a time. Therefore, only one image can be bound to this display at a given time.

imageRGB32

```
const ccPelBuffer_const<ccPackedRGB32Pel> imageRGB32()
  const;
```

Returns a reference to the 32-bit color image currently in the display.

Notes

The display can manage only one image at a time. Therefore, only one image can be bound to this display at a given time.

enableOverlay

```
virtual overlayState enableOverlay() const;
virtual void enableOverlay(overlayState);
```

- `virtual overlayState enableOverlay() const;`

Returns the state of the overlay layer (*eOverlayPlane* if enabled, or *eDisableOverlayPlane* if disabled). Should return *eNone* if both the overlay layer and chromakeying are disabled.

Throws

ccUIError::NotImplemented
The overlay layer is not supported in the derived class.

- `virtual void enableOverlay(overlayState state);`

Enables or disables the overlay layer of the drawing tablet (*ccUITablet::eOverlayLayer*) on host-based displays (**ccDisplayConsole** or **ccWin32Display**).

Parameters

state The state of the overlay layer. Must be one of:

ccDisplay::eOverlayPlane (for enabled),
ccDisplay::eDisableOverlayPlane (for disabled)

Notes

Never pass *ccDisplay::eNone* as an argument to this method.

Before you can draw into the overlay layer on a host-based display (**ccDisplayConsole** or **ccWin32Display**), you must enable the overlay layer by calling **enableOverlay(ccDisplay::eOverlayPlane)**.

Before enabling the overlay layer, call **ccUITablet::overlaySupported()** to check whether the overlay layer is supported on a particular platform.

Attempting to enable the overlay layer on a display that does not support it, or on an embedded display, has no effect. In the first case, the overlay layer is not supported. In the second, the overlay layer is already enabled.

Before disabling the overlay layer on a host-based display, remove any interactive shapes from the overlay layer and erase all sketches on all drawing layers.

Throws

ccUIError::NotImplemented
The overlay layer is not supported in the derived class.

canBlink

```
virtual bool canBlink() const;
```

Returns true if the derived display object supports blinking colors, false otherwise.

Notes

Blinking colors are supported only on embedded displays, and only in the overlay layer.

enableBlink

```
virtual bool enableBlink();
```

Starts the blinking of the color specified by **blinkColor()**. Returns true if blinking started successfully, false otherwise.

Notes

Blinking colors are supported only on embedded displays, and only in the overlay layer.

disableBlink

```
virtual bool disableBlink();
```

Stops the blinking started by **enableBlink()**. Returns true if blinking was successfully halted, false if blinking was not enabled or could not be halted.

Notes

Blinking colors are supported only on embedded displays, and only in the overlay layer.

blinkRate

```
virtual void blinkRate(double seconds);

virtual double blinkRate() const;
```

Notes

Blinking colors are supported only on embedded displays, and only in the overlay layer.

- `virtual void blinkRate(double seconds);`

Sets the blink rate. The blink rate must be set prior to calling **enableBlink()**, otherwise the previous blink rate is used.

Parameters

seconds The blink rate in seconds. Must be greater than zero.

- `virtual double blinkRate() const;`

  Returns the current blink rate.

blinkColor

```
virtual void blinkColor(
  const cmStd vector<ccColor>& blink);

virtual const cmStd vector<ccColor>& blinkColor();
```

Notes

Blinking colors are supported only on embedded displays, and only in the overlay layer.

- ```
  virtual void blinkColor(
    const cmStd vector<ccColor>& blink);
  ```

  Specifies the colors that blink in the overlay plane when **enableBlink()** is called.

  Parameters

  blink A vector of colors. See *color.h* for a list of predefined colors.

  Notes

  Once blinking is enabled, objects drawn in the colors specified in the *blink* parameter blink against a black background.

- `virtual const cmStd vector<ccColor>& blinkColor();`

  Returns a vector of the colors that blink when **enableBlink()** is called.
  ```

startLiveDisplay

```
virtual bool startLiveDisplay(ccAcqFifo* fifo,
 ccLiveDisplayProps* p = 0);

virtual bool startLiveDisplay(ccAcqFifoPtrh fifo,
 ccLiveDisplayProps* p = 0);
```

Starts continuous live display of images acquired through the specified acquisition FIFO. While live display is in progress, **startLiveDisplay()** owns the FIFO. Your application should not make changes to the FIFO until you stop live display.

All overloads return true if live display is able to start successfully, false otherwise.

You can set live display properties, such as whether to use a different transform than the transform associated with each pel buffer acquired for live display, by assigning them to a **ccLiveDisplayProps** object. You can pass this object as the optional second argument to **startLiveDisplay()**. See **ccLiveDisplayProps** for details.

Some derived classes do not support live display.

Notes

Repeated calls to **startLiveDisplay()**, even with a different FIFO, have no effect once live display has already started. Before you can start live display on a different FIFO, you must first call **stopLiveDisplay()**.

On most frame grabbers, the acquisition rate is maximized for the supplied acquisition FIFO and the current frame grabber. On some frame grabbers, you can reduce the acquisition frame rate to account for hardware limitations that may affect the refresh rate. To control the acquisition and display frame rate, call **ccLiveDisplayProps::restartDelay()**, passing a delay amount as an argument.

Call **fifo->prepare(0.0)** before calling **startLiveDisplay()** to reduce the delay between when the first image is acquired and when it appears in the display.

There is a finite amount of time between when **startLiveDisplay()** returns and when the live display actually starts.

-
```
virtual bool startLiveDisplay(ccAcqFifo* fifo,
 ccLiveDisplayProps* p = 0);
```

Starts continuous live display of images acquired through the supplied FIFO.

Parameters

fifo	Pointer to the acquisition FIFO to use for live display. The FIFO must be configured to use the manual trigger model.
p	The live display properties. See **ccLiveDisplayProps** for more information.

Throws

ccDisplay::BadFifo
The input FIFO is not valid.

- `bool startLiveDisplay(ccAcqFifoPtrh fifo, ccLiveDisplayProps* p = 0);`

Starts continuous live display of images acquired through the supplied FIFO.

Parameters

fifo Pointer handle to the acquisition FIFO to use for live display. The FIFO must be configured to use the manual trigger model.

p The live display properties. See **ccLiveDisplayProps** for more information.

Throws

ccDisplay::BadFifo
The input FIFO is not valid.

stopLiveDisplay

```
virtual void stopLiveDisplay();
```

Stops live display.

Notes

You must call **stopLiveDisplay()** before modifying any FIFO properties, or switching the FIFO that is supplying the images.

Repeated calls to **stopLiveDisplay()** have no effect once live display is inactive.

isLiveEnabled

```
virtual bool isLiveEnabled() const;
```

Returns true if live display has started and is active, false otherwise.

waitForVerticalBlank

```
virtual void waitForVerticalBlank(bool set);
```

Sets the *waitForVerticalBlank* flag. If this flag is true, the display is synchronized to the vertical blank refresh of the monitor. If false, the display is not synchronized.

Parameters

set The new value for the *waitForVerticalBlank* flag.

Notes

Be aware that not all display drivers support this synchronization feature. For those that do, execution time may increase when *waitForVerticalBlank* is true.

getDisplayedImage

```
virtual void getDisplayedImage(
  ccPelBuffer<c_UInt8>& buf,
  ccUITablet::Layers layer = ccUITablet::eUnspecifiedLayer)
    = 0;

virtual void getDisplayedImage(
  ccPelBuffer<ccPackedRGB16Pel>& buf,
  ccUITablet::Layers layer = ccUITablet::eUnspecifiedLayer)
    = 0;

virtual void getDisplayedImage(
  ccPelBuffer<ccPackedRGB32Pel>& buf,
  ccUITablet::Layers layer = ccUITablet::eUnspecifiedLayer)
    = 0;
```

Returns the image, graphics, and background as currently displayed on the specified layer in the viewable client area of the display window. If no layer is specified, the returned pel buffer contains the combined pixel data from the image and overlay layers.

The return pel buffer must be allocated and sized to the current size of the client area of the display window. This can be done, for example, as follows:

```
ccPelBuffer<c_UInt8> buf(display.clientArea().x(),
                         display.clientArea().y());
display.getDisplayedImage(buf);
```

Notes

Always use display coordinates (*eDisplayCoords*) to allocate and size the pel buffer used with this method.

To achieve best performance, use the overload that matches the current desktop depth. If the desktop depth differs from the destination pel buffer, data conversion will occur with possible loss of data.

Retrieval of the displayed image always starts from the top left corner of the display window, and does not take into account any offset that may be set in the *buf* return parameter. To obtain a smaller portion of the displayed window, capture the entire window and then use subwindowing.

- ```
  virtual void getDisplayedImage(
    ccPelBuffer<c_UInt8>& buf,
    ccUITablet::Layers layer = ccUITablet::eUnspecifiedLayer)
     = 0;
  ```

  Parameters

  buf — The 8-bit pel buffer in which pixel data are returned.

  layer — The layer (default = unspecified). If no layer is specified, the returned pel buffer contains the combined pixel data from the image and overlay layers.

- ```
  virtual void getDisplayedImage(
    ccPelBuffer<ccPackedRGB16Pel>& buf,
    ccUITablet::Layers layer = ccUITablet::eUnspecifiedLayer)
     = 0;
  ```

  Parameters

  buf — The 16-bit packed RGB pel buffer in which pixel data are returned.

  layer — The layer (default = unspecified). If no layer is specified, the returned pel buffer contains the combined pixel data from the image and overlay layers.

- ```
  virtual void getDisplayedImage(
    ccPelBuffer<ccPackedRGB32Pel>& buf,
    ccUITablet::Layers layer = ccUITablet::eUnspecifiedLayer)
     = 0;
  ```

  Parameters

  buf — The 32-bit packed RGB pel buffer in which pixel data from the specified layer are returned.

  layer — The layer (default = unspecified). If no layer is specified, the returned pel buffer contains the combined pixel data from the image and overlay layers.

liveFrameRate

```
virtual double liveFrameRate() const;
```

Returns the live display frame rate in frames per second.
```

Protected Member Functions

panChanged

```
virtual void panChanged();
```

Called by **pan()** to notify that a new pan setting has been applied to the display.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when the image pan changes. Unless overridden, this function performs no action and simply returns.

magChanged

```
virtual void magChanged();
```

Called by **mag()** to notify that a new magnification setting has been applied to the display.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when the magnification changes. Unless overridden, this function performs no action and simply returns.

imageChanged

```
virtual void imageChanged();
```

Called by **image()** to notify that a new image has been displayed.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when the image changes. Unless overridden, this function performs no action and simply returns.

mouseModeChanged

```
virtual void mouseModeChanged();
```

Called by **mouseMode()** to notify that a new mouse mode has been applied to the display.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when the mouse mode changes. Unless overridden, this function performs no action and simply returns.

colorMapChanged

```
virtual void colorMapChanged();
```

Called by **colorMap()** to notify that a new color map for the 8-bit desktop depth has been applied to the display.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when the color map changes. Unless overridden, this function performs no action and simply returns.

imageMapChanged

```
virtual void imageMapChanged();
```

Called by **colorMapEx()** to notify that a new color map for non-8-bit desktop depths has been applied to the display.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when the color map for non-8-bit desktops changes. Unless overridden, this function performs no action and simply returns.

redraw_

```
virtual void redraw_();
```

Implements drawing of the image and all static graphics.

Notes

Overrides the virtual method inherited from the **ccUITablet** base class.

dragStart_

```
virtual void dragStart_ (const ccIPair& startPos);
```

Implements panning of the image. Called at the start of a pan operation.

Parameters

startPos The mouse starting position for the drag.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

dragAnimate_

```
virtual void dragAnimate_(const ccIPair&, const ccIPair&);
```

Implements panning of the image. Called repeatedly during a pan operation to implement visual dragging of the image on the screen. Graphics are not animated during a pan operation, but they are repositioned when dragging stops.

Parameters

ccIPair The starting image position, in screen coordinates.

ccIPair The new image position in screen coordinates.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

dragStop_ `virtual void dragStop_ (const ccIPair&, const ccIPair&);`

Implements panning of the image. Called at the end of a pan operation.

Parameters

ccIPair The starting image position, in screen coordinates.

ccIPair The ending image position in screen coordinates.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

resize_ `virtual void resize_();`

Implements resizing of the display window. Adjusts the child frame scope rectangles and maintains the pan.

Notes

Overrides the virtual method inherited from the **ccUITablet** base class.

This method triggers an **updatePanRanges()** event to notify that the window has been resized and **defaultPan()** is true.

click_ `virtual void click_();`

Implements mouse mode behavior when the user clicks with the mouse on an image.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

dblClick_ `virtual void dblClick_();`

Called when a double click occurs while the mouse pointer is in the image. Scrolls the double-clicked point to the center of the display if the display is draggable.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

keyboard_ `virtual void keyboard_(const ccKeyboardEvent &ev);`

Changes the mouse mode if a <Ctrl> key is pressed or released.

Parameters

ev The keyboard event.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

mouseEnter_ `virtual void mouseEnter_();`

Sets the mouse mode. Called when the mouse pointer enters the image with the left mouse button depressed.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

idleMouseEnter_

`virtual void idleMouseEnter_();`

Sets the mouse mode. Called when the mouse pointer enters the image with no mouse button depressed.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

mouseLeave_ `virtual void mouseLeave_();`

Restores the cursor to the state it was in before entering the image. Called when the mouse pointer leaves the image.

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

mouseRight_

```
virtual void mouseRight_(const ccIPair& p, MouseAction m,
  c_UInt32 keys);
```

Called when the right-hand mouse button is depressed while the mouse pointer is in the image. Implements right-click for *eZoomIn* and *eZoomOut* modes.

Parameters

p	The last mouse position, in screen coordinates.
m	Type of mouse event that caused this call. Must be one of the **ccMouseEvent::Event** enums (*eNone, eMovement, eDownClick, eDoubleClick, eUpClick, eDwell*).
keys	Modifier keys depressed when the mouse action occurred. Must be one of the **ccMouseEvent::Key** enums (*eNoKey*, *eAlt*, *eControl*, *eShift*).

Notes

Overrides the virtual method inherited from the **ccUIObject** base class.

updateDisplay

```
void updateDisplay(Layers lyr = eNumLayers);
```

Updates the display by calling **ccUITablet::update()** on the specified layer.

Parameters

lyr The layer to update.

Notes

If **drawingDisabled()** is true, no update occurs.

updatePanRanges

```
virtual void updatePanRanges();
```

Called when the minimum or maximum pan range changes. This can result from calls to **image()**, **minPan()**, **maxPan()**, **image()**, or **resize_()**.

Notes

You can override this callback function in a derived class of your own to take application-specific actions when either of the pan ranges changes. Unless overridden, this function performs no action and simply returns.

Deprecated Members

selectArea

```
ccPelRect selectArea(
  CoordinateSystem cs,
  const TCHAR* prompt,
  const ccColor &c = ccColor::greenColor());
```

For new development, use the overload on page 1216.

selectPoint

```
ccDPair selectPoint(CoordinateSystem cs,
  const TCHAR* prompt,
  const ccColor& c = ccColor::greenColor());
```

For new development, use the overload on page 1218.

ccDisplay

ccDisplayConsole

```
#include <ch_cvl/windisp.h>

class ccDisplayConsole : public CFrameWnd, public ccWin32Display;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	No

The **ccDisplayConsole** class is based on the Microsoft Foundation Class (MFC) frame window. Besides a frame, it contains a client window and optionally displays a tool bar, status bar, and scroll bars. The image is displayed in the client area of the window, which is resized within the frame window. The status bar displays image coordinates, pixel values, and an optional status message. The tool bar contains buttons that zoom in and out, fit, or pan the image, and enable or disable grid lines.

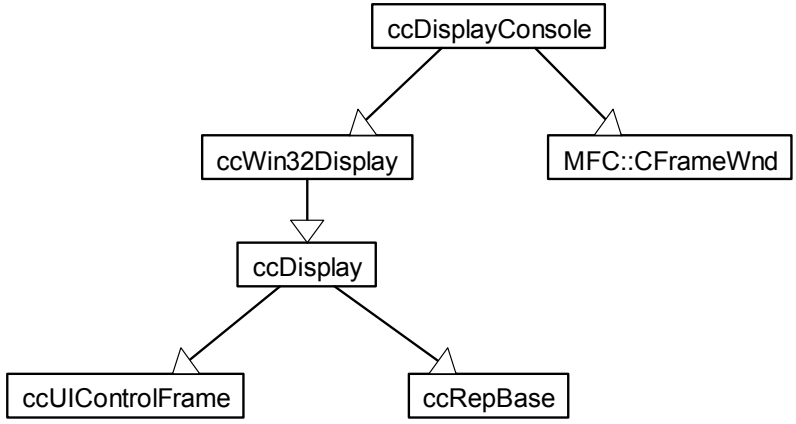


Figure 1. ccDisplayConsole class inheritance hierarchy

This class is derivable. In other words, you can create your own subclass that derives directly from **ccDisplayConsole**, implementing the virtual methods it inherits from the **ccDisplay** class hierarchy. You can also derive your own class to add more Windows MFC message map methods from the MFC class hierarchy.

This class must not be constructed at static initialization time (STI). It may be allocated either on the heap or on the stack as a local variable. Cognex recommends that a **ccDisplayConsole** be allocated on the heap when the close window action is set to *ccDisplayConsole::eCloseDelete*. If the display console contains an image pel buffer that was acquired from an acquisition FIFO, first delete the FIFO before deleting the display console. See **closeAction()** for more information.

ccDisplayConsole uses resources contained within the DLL (*cogdisp*.dll*) distributed with CVL. These resources are automatically loaded either upon construction, or as needed. If the distributed DLL is repackaged inside of another regular (non-ActiveX) MFC DLL, you must call **cfInitializeDisplayResources()** before constructing a **ccDisplayConsole** object to tell it where to find the display resources it needs.

If you are not programming under MFC, use the **ccWin32Display** class instead. This class uses the Microsoft Win32 API.

The **ccDisplayConsole** display class has the following capabilities:

- Displaying images (pel buffers)
- Displaying live images from Cognex frame grabbers
- Adding static graphics to the display
- Adding interactive graphics to the display
- GUI with mouse support

Constructors/Destructors

ccDisplayConsole

```
ccDisplayConsole(const ccIPair& windowSize,
  const TCHAR* title = cmT("Display Console"),
  CWnd* parentWnd = NULL);

ccDisplayConsole(const TCHAR* title, const ccIPair& ul,
  const ccIPair& windowSize,
  bool makeVisible = true, bool minimized = false,
  CWnd* parentWnd = NULL);

virtual ~ccDisplayConsole();
```

Both constructors construct fully-functional, MFC-based, framed windows that can display images (pel buffers) and CVL static and interactive graphics.

The **ccDisplayConsole** constructors are not available at static initialization time.

Only static and interactive graphics from the CVL API can be added to a **ccDisplayConsole**. This object cannot manage user-created graphics from Win32/GDI or any other graphics package.

A **ccDisplayConsole** manages refresh updates of the display window automatically whenever the image or graphics change.

Construct **ccDisplayConsole** objects only in a thread or application that contains a Windows message handler so that window events, such as mouse movement, are properly retrieved and dispatched. For an example of how to do this, see the display sample code.

- ```
  ccDisplayConsole(const ccIPair& windowSize,
    const TCHAR* title = cmT("Display Console"),
    CWnd* parentWnd = NULL);
  ```

  This constructor sets the client area size to the specified size, and centers the framed window on the desktop. This is the simplest way to create a display console.

  Parameters

windowSize	The size of the client area of the display window in pixels. The actual framed window is larger than *windowSize*, as it includes the scroll bars, status bar, and title bar.
title	The title of the window (default = “Display Console”).
parentWnd	The parent window, if any. This window is passed on to **CFrameWnd::Create()** during construction of the frame window.
  ```

Throws

ccUIError::InitFailed

The object failed to construct. This may indicate a failure to properly load the MFC display resource file, or a conflict with other resource files.

- ```
  ccDisplayConsole(const TCHAR* title,
    const ccIPair& ul, const ccIPair& windowSize,
    bool makeVisible = true, bool minimized = false,
    CWnd* parentWnd = NULL);
  ```

  This constructor gives you more control over window placement and appearance, allowing you to specify its location, whether the window is visible or hidden, and whether it is minimized or full size.

  Parameters

  title — The title of the window (default = “Display Console”).

  ul — The location of the upper left corner of the window in screen coordinates.

  windowSize — The size of the client area of the display window in pixels. The actual framed window is larger than *windowSize*, as it includes the scroll bars, status bar, and title bar.

  makeVisible — True makes the window initially visible upon construction. You can also use the MFC method **CWnd::ShowWindow()** to control the visibility of the window.

  minimized — True makes the window initially minimized upon construction. You can also use the MFC method **CWnd::ShowWindow()** to control the active state of the window.

  parentWnd — The parent window, if any. This window is passed on to **CFrameWnd::Create()** during construction of the frame window.

  Throws

  ccUIError::InitFailed

  The object failed to construct. This may indicate a failure to properly load the MFC display resource file, or a conflict with other resource files.

- ```
  virtual ~ccDisplayConsole();
  ```

  Destroys the display console.

Notes

Live display is automatically stopped before the window is destroyed.

Enumerations

CloseAction

```
enum CloseAction;
```

This enumeration specifies the action to be performed when a **ccDisplayConsole** window is closed, using either the **Close** (X) box or the menu item.

Value	Meaning
eCloseDisabled	Disables the close option of the display console. In other words, this option disables the **Close** button (or menu item) so that nothing happens when the user tries to select it.
eCloseHide	Hides the display console when the **Close** button (or menu item) is selected, but does not delete the **ccDisplay** object. To make a **ccDisplayConsole** window visible again after it has been hidden, call the MFC function **ShowWindow()** as follows: `display->ShowWindow(SW_SHOWNOACTIVATE);`
eCloseDelete	Deletes the **ccDisplayConsole** object when the window is closed. This has the effect of both disabling and hiding the display console. The **ccDisplayConsole** must be created on the heap to use the *eCloseDelete* option.

Operators

operator new

```
void* operator new(size_t sz);
```

Allocates storage on the heap for a new **ccDisplayConsole**, initializes it, and returns a pointer to it. Disambiguates the **new()** operator.

Parameters

sz — A non-negative, integral value representing the number of bytes to allocate. If 0, this operator returns a NULL pointer.

Notes

Both the **ccUIObject** class, defined in *<ch_cvl/uifrmwrk.h>*, and the **CObject** class, defined in MFC, override **new()**. This operator forces the call to the MFC **CObject** version. This avoids the use of the CVL **ccUIObject** version, which sets **ccUIObject::autoDelete()** to true.

operator delete

```
void operator delete(void* ptr);
```

Deallocates the storage for the **ccDisplayConsole** pointed to by *ptr*.

Public Member Functions

showStatusBar

```
void showStatusBar(bool vis);
```

Shows or hides the display console's status bar.

Parameters

vis	True shows the status bar, false hides it.

showToolBar

```
void showToolBar(bool vis);
```

Shows or hides the display console's tool bar.

Parameters

vis	True shows the tool bar, false hides it.

statusBarText

```
void statusBarText(const ccCvlString &text,
     bool resize=true);
```

Set the text displayed in the status bar. By default, the status bar is resized to fit the supplied text.

Parameters

text	The text of the status bar.
resize	If true, the bar is resized to fit the supplied text.

Notes

The status bar can be changed only from the thread that created this object. Attempting to set the status bar text from another thread has no effect.

closeAction

```
void closeAction (CloseAction action);
CloseAction closeAction() const;
```

- `void closeAction (CloseAction action);`

  Sets the action to perform when the user clicks on the window's **Close** (X) button or selects the **Close** menu item.

  Parameters

  action — The close action to perform. Must be one of the following:

  ccDisplayConsole::eCloseDisabled (default)
  ccDisplayConsole::eCloseHide
  ccDisplayConsole::eCloseDelete

  Notes

  The standard **CFrameWnd** behavior is *ccDisplayConsole::eCloseDelete*. Select this action if the **ccDisplayConsole** window is the application's main frame window. The **ccDisplayConsole** must be created on the heap to use this option.

- `CloseAction closeAction() const;`

  Returns the current close action.

RecalcLayout

```
virtual void RecalcLayout(BOOL bNotify = TRUE);
```

This function is the CVL implementation of the MFC **CFrameWnd::RecalcLayout()** method. If you add additional panes (such as status bars) to a class that you derive from **ccDisplayConsole**, you should override this function to position your pane after a window has been resized. See the MFC documentation for more information.

ccDisplayConsole

ccEdgelet

```
#include <ch_cvl/edge.h>

class ccEdgelet: public ccEdgeletBase;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class describes a single sub-pixel edge in an image. You should not construct **ccEdgelet** objects, nor should you attempt to modify them.

A **ccEdgelet** object stores the whole-pixel location of the edge using a pair of 16-bit integer values. For this reason, a **ccEdgelet** object cannot represent an edge that has an image offset of greater than +32767 or less than -32768. If you are working with extremely large images, you can specify the use of the **ccEdgelet2** class, described on page 1245, to represent edges, or you can specify that the tool use multiple blocks of memory with offset specifiers, as described in the function **ccEdgeletParams::**edgeTypeRequest() on page 1279

For information on using the edge tool with very large images, see the description of the class **ccEdgeletSet** on page 1281.

Constructors/Destructors

ccEdgelet

```
ccEdgelet();

ccEdgelet(const cc2Vect& position, c_UInt16 magnitude,
  const ccAngle8& angle);
```

- `ccEdgelet();`

- `ccEdgelet(const cc2Vect& position, c_UInt16 magnitude, const ccAngle8& angle);`

  Constructs a **ccEdgelet** with the specified position, magnitude and angle.

  Parameters

position	The sub-pixel location
magnitude	The magnitude, expressed as a grey-level change.

angle The angle.

Notes

The constructed **ccEdgelet** will store the position as a one-dimensional offset from a pixel center along a line drawn through the pixel center at the edge angle.

Public Member Functions

position

```
cc2Vect position() const;
void position(const cc2Vect& pos);
```

- `cc2Vect position() const;`

Returns the sub-pixel location of this **ccEdgelet**.

- `void position(const cc2Vect& pos);`

Sets the sub-pixel location of this **ccEdgelet**.

Parameters

pos The position to set.

Notes

The **ccEdgelet** stores the position as a one-dimensional offset from a pixel center along a line drawn through the pixel center at the edge angle. The value returned by **position()** may not be exactly the same as the position specified upon construction or that passed as an argument to **position()**.

posQuickX

```
double posQuickX(void) const;
```

Returns the whole-pixel x-axis location of this **ccEdgelet** with the lowest possible overhead, assuming that the edge location has been determined.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

posQuickY

```
double posQuickY(void) const;
```

Returns the whole-pixel y-axis location of this **ccEdgelet** with the lowest possible overhead, assuming that the edge location has been determined.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

posQuickR

```
double posQuickR(void) const;
```

Returns the sub-pixel offset of this **ccEdgelet** with the lowest possible overhead, assuming that the edge location has been determined.

This is the distance, in image coordinate system units, from the pixel center to the sub-pixel edge, measured along a line drawn through the pixel center in the direction of the edge.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

magnitude

```
c_UInt16 magnitude() const;
void magnitude(c_UInt16 mag);
```

- `c_UInt16 magnitude() const;`

  Returns the magnitude of this **ccEdgelet**.

- `void magnitude(c_UInt16 mag);`

  Sets the magnitude of this **ccEdgelet**.

  Parameters

  mag The edge magnitude to set

angle

```
ccAngle8 angle() const;
void angle(const ccAngle8& ang);
```

- `ccAngle8 angle() const;`

  Returns the angle of this **ccEdgelet**, relative to the image coordinate system x-axis.

- `void angle(const ccAngle8& ang);`

  Sets the angle of this **ccEdgelet**, relative to the image coordinate system x-axis.

Parameters

ang The angle to set

gradient

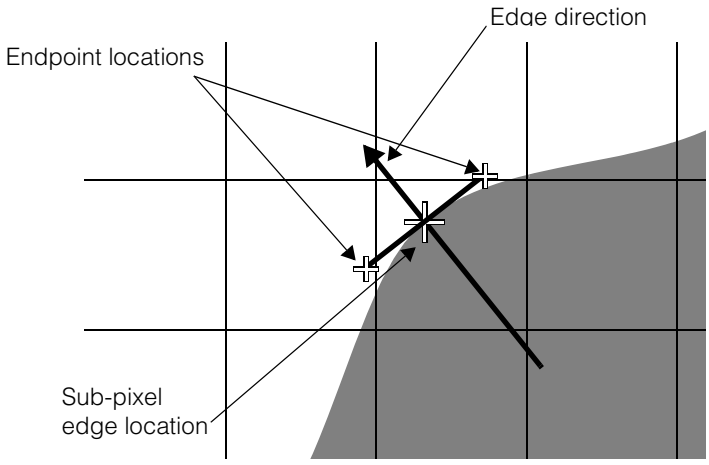
```
cc2Vect gradient() const;
```

Returns a unit vector in positive gradient direction (dark-to-light) that describes this **ccEdgelet**.

endpoints

```
void endpoints(cc2Vect& left, cc2Vect& right,
  bool useSubpixel = true) const;
```

Returns the location of a one-pixel-long line perpendicular to this **ccEdgelet**'s edge direction and drawn through the sub-pixel edge location, as shown in the following figure:



Parameters

left A reference to the left endpoint of the line

right A reference to the right endpoint of the line

useSubpixel If true, locate the line with sub-pixel accuracy; if false, locate the line through the pixel center.

Notes

This function is intended to support the graphic display of **ccEdgelet**s.

ccEdgelet2

```
#include <ch_cvl/edge.h>

class ccEdgelet2;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class describes a single sub-pixel edge in an image. You should not construct **ccEdgelet2** objects, nor should you attempt to modify them.

A **ccEdgelet2** object stores the whole-pixel location of the edge using a pair of 32-bit integer values. Unlike a **ccEdgelet** object, a **ccEdgelet2** object can represent edges in images of any size or offset.

For information on using the edge tool with very large images, see the description of the class **ccEdgeletSet** on page 1281.

Constructors/Destructors

ccEdgelet2

```
ccEdgelet2();

ccEdgelet2(const cc2Vect& position, c_UInt16 magnitude,
   const ccAngle8& angle);
```

- `ccEdgelet2() {}`

- `ccEdgelet2(const cc2Vect& position, c_UInt16 magnitude, const ccAngle8& angle);`

  Constructs a **ccEdgelet2** with the specified position, magnitude and angle.

  Parameters

position	The sub-pixel location
magnitude	The magnitude, expressed as a grey-level change.
angle	The angle.

Notes

The constructed **ccEdgelet2** will store the position as a one-dimensional offset from a pixel center along a line drawn through the pixel center at the edge angle.

Public Member Functions

position

```
cc2Vect position() const;
void position(const cc2Vect& pos);
```

- `cc2Vect position() const;`

Returns the sub-pixel location of this **ccEdgelet2**.

- `void position(const cc2Vect& pos);`

Sets the sub-pixel location of this **ccEdgelet2**.

Parameters

pos The position to set.

Notes

The **ccEdgelet2** stores the position as a one-dimensional offset from a pixel center along a line drawn through the pixel center at the edge angle. The value returned by **position()** may not be exactly the same as the position specified upon construction or that passed as an argument to **position()**.

posQuickX

```
double posQuickX(void) const;
```

Returns the whole-pixel x-axis location of this **ccEdgelet2** with the lowest possible overhead, assuming that the edge location has been determined.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

posQuickY

```
double posQuickY(void) const;
```

Returns the whole-pixel y-axis location of this **ccEdgelet2** with the lowest possible overhead, assuming that the edge location has been determined.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

posQuickR

```
double posQuickR(void) const;
```

Returns the sub-pixel offset of this **ccEdgelet2** with the lowest possible overhead, assuming that the edge location has been determined.

This is the distance, in image coordinate system units, from the pixel center to the sub-pixel edge, measured along a line drawn through the pixel center in the direction of the edge.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

---

magnitude

```
c_UInt16 magnitude() const;
```

```
void magnitude(c_UInt16 mag);
```

- ```
  c_UInt16 magnitude() const;
  ```

  Returns the magnitude of this **ccEdgelet2**.

- ```
  void magnitude(c_UInt16 mag);
  ```

  Sets the magnitude of this **ccEdgelet2**.

  Parameters

  mag The edge magnitude to set

---

angle

```
ccAngle8 angle() const;
```

```
void angle(const ccAngle8& ang);
```

- ```
  ccAngle8 angle() const;
  ```

  Returns the angle of this **ccEdgelet2**, relative to the image coordinate system x-axis.

- ```
  void angle(const ccAngle8& ang);
  ```

  Sets the angle of this **ccEdgelet2**, relative to the image coordinate system x-axis.

  Parameters

  ang The angle to set

gradient

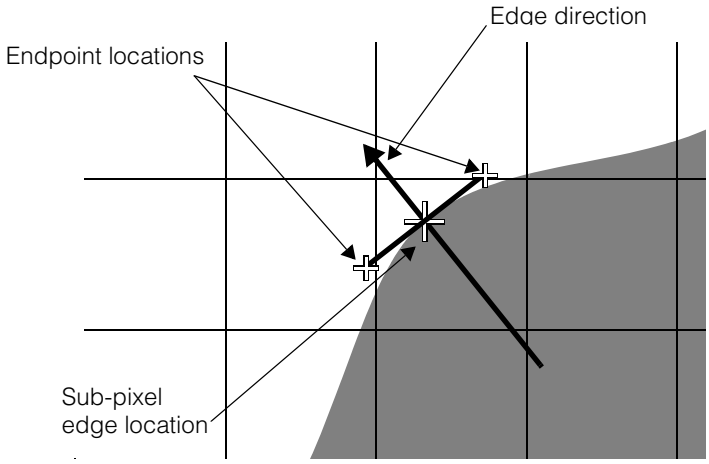
```
cc2Vect gradient() const;
```

Returns a unit vector in positive gradient direction (dark-to-light) that describes this **ccEdgelet2**.

endpoints

```
void endpoints(cc2Vect& left, cc2Vect& right,
  bool useSubpixel = true) const;
```

Returns the location of a one-pixel-long line perpendicular to this **ccEdgelet2**'s edge direction and drawn through the sub-pixel edge location, as shown in the following figure:



Parameters

left	A reference to the left endpoint of the line
right	A reference to the right endpoint of the line
useSubpixel	If true, locate the line with sub-pixel accuracy; if false, locate the line through the pixel center.

Notes

This function is intended to support the graphic display of **ccEdgelet2**s.

ccEdgeletChainFilter

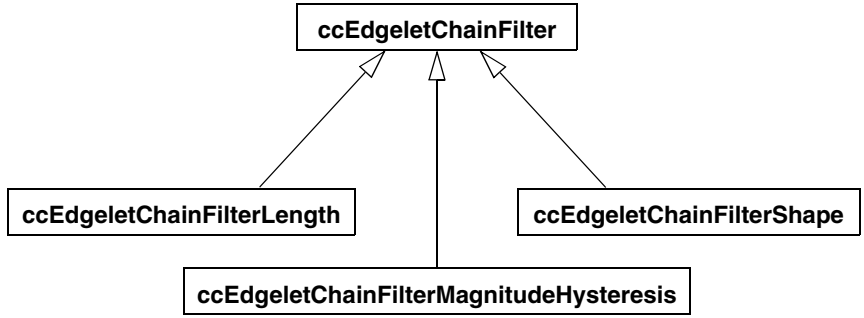
```
#include <ch_cvl/edgefilt.h>

class ccEdgeletChainFilter: public virtual ccPersistent,
                            public virtual ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This is the abstract base class for derived edgelet chain filter classes. Derived classes should override the virtual member function **filter()** and should implement a particular filtering algorithm. The three filters provided by Cognex are shown below.



Edgelet chain filters are designed to filter edgelet chains produced by the edge detection tool (**cfEdgeDetect()**). You first run the edge detection tool to find a set of edgelets in an image, and then run a filter tool on the detected edgelets to eliminate edgelets you do not want to consider in your application.

The edge detection tool produces a vector of edgelets, and an index chain list that describes how these edgelets are grouped as chains. The edgelet filter tool creates a new edgelet chain list that eliminates unwanted edgelets but still refers to the input edgelet vector. The input edgelet vector is not modified. After running the filter the edgelet vector is unchanged and contains both wanted and unwanted edgelets, however only the wanted edgelets are described in the output filtered chain list.

Constructors/Destructors

ccEdgeletChainFilter

```
ccEdgeletChainFilter();

virtual ~ccEdgeletChainFilter();
```

- `ccEdgeletChainFilter();`

  Default constructor.

- `virtual ~ccEdgeletChainFilter();`

  Destructor.

Public Member Functions

filter

```
virtual void filter(
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const ccIndexChainList &inputChains,
  ccIndexChainList &filteredChains) const = 0;
```

This function must be overridden in a derived class.

Filters the given edgelets based on the criteria implemented in derived classes. The filtering algorithm analyzes the input edgelets and applies the filtering criteria specified in the derived class. Edgelets that meet the filter criteria are saved and the other edgelets are discarded (filtered out). The surviving edgelets are described by *filteredChains* which contains chains of surviving edgelets in the *inputEdgelets* vector.

A new **ccIndexChainList** is created using the surviving edgelets by eliminating chains, trimming chains by eliminating edgelets at either end, or by breaking chains into multiple chains. If a chain is broken by eliminating an interior edgelet, the two resulting chains cannot be joined.

Parameters

inputEdgelets	A vector of edgelets to be filtered.
inputChains	The input index chain list for *inputEdgelets*.
filteredChains	The new index chain list produced when you execute this function.

Throws

ccEdgeletChainFilterDefs::OutOfBounds

If the input chains have any out of bounds pointers to edgelets or edgelet indices.

Notes

The filtering will be performed correctly even if *inputChains* and *filteredChains* are references to the same vector. The output chain list overwrites the input chain list.

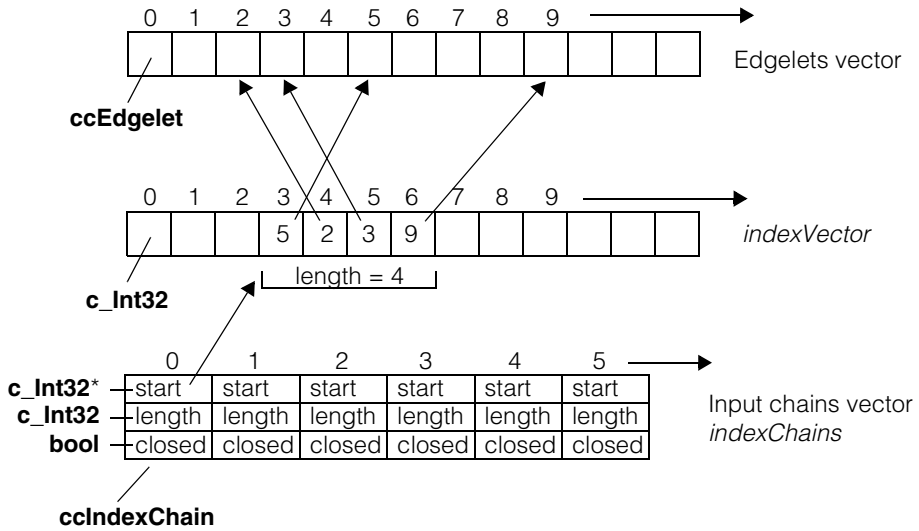
Any existing elements in *filteredChains* will be discarded before the filtering starts.

Any derived class overriding this function should replicate the behavior described in these notes.

makeChainList

```
static void makeChainList(
  const cmStd vector<ccIndexChain> &indexChains,
  const cmStd vector<c_Int32> &indexVector,
  ccIndexChainList &chainList);
```

Convert a vector of **ccIndexChain** objects to a **ccIndexChainList** representation. *indexChains* is a vector **ccIndexChain** objects, each representing a chain of **ccEdgelets**. Each **ccIndexChain** object contains a pointer into *indexVector* which holds indices into a vector of **ccEdgelets**. See the following diagram. The resulting *chainList* can be used as an input to a filtering operation along with the vector of **ccEdgelets**.



The *indexChains* vector is converted into an equivalent **ccIndexChainList** object.

Parameters

indexChains A vector of input chains to be converted.

indexVector The index vector associated with *indexChains*.

chainList The index chain list created.

Throws

ccEdgeletChainFilterDefs::OutOfBounds
If *indexChains* or *chainList* has any out of bounds pointers to *indexVector*.

Notes

chainList holds a deep copy of *indexVector* and *indexChains*.

makeChainVect

```
static void makeChainVect(
  const ccIndexChainList &chainList,
  cmStd vector<c_Int32> &indexVector,
  cmStd vector<ccIndexChain> &indexChains);
```

Convert a **ccIndexChainList** object into a vector of **ccIndexChain** objects. *chainList* contains **ccIndexChain** objects, each representing a chain of **ccEdgelets**. Each **ccIndexChain** object contains a pointer into *indexVector* which holds indices into a vector of **ccEdgelets**.

This function performs the inverse of **makeChainList()** above. *chainList* is converted into an equivalent vector of **ccIndexChain** objects. See the above diagram.

Parameters

chainList The chain list to be converted.

indexVector The index vector associated with *chainList*.

indexChains The index chains vector created.

Throws

ccEdgeletChainFilterDefs::OutOfBounds
If *indexChains* or *chainList* has any out of bounds pointers to *indexVector*.

Typedefs

ccEdgeletChainFilterPtrh

```
typedef ccPtrHandle<ccEdgeletChainFilter>
                                        ccEdgeletChainFilterPtrh;
```

ccEdgeletChainFilterPtrh_const

```
typedef ccPtrHandle_const<ccEdgeletChainFilter>
                                  ccEdgeletChainFilterPtrh_const;
```

ccEdgeletChainFilter

ccEdgeletChainFilterLength

```
#include <ch_cvl/edgefilt.h>

class ccEdgeletChainFilterLength: public ccEdgeletChainFilter;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This is a filter class derived from **ccEdgeletChainFilter**. This filter provides functionality for filtering **ccEdgelet** chains based on their length, regardless of the open or closed status of the chains.

Please see the **ccEdgeletChainFilter** base class reference page for additional information about using this class.

Constructors/Destructors

ccEdgeletChainFilterLength

```
explicit ccEdgeletChainFilterLength(
  c_Int32 minChainLength = 1);
```

Constructor.

Parameters

minChainLength

The minimum chain length. Must be at least 1.

Throws

ccEdgeletChainFilterDefs::BadParams

If *minChainLength* < 1.

Operators

operator==

```
bool operator== (
  const ccEdgeletChainFilterLength &that) const;
```

Returns true if the internal data members of this object are exactly equal to the corresponding data members of *that*. Returns false otherwise.

Parameters

that The **ccEdgeletChainFilterLength** object to compare with this object.

operator!=

```
bool operator!= (
  const ccEdgeletChainFilterLength &that) const;
```

Returns true if the internal data members of this object are not exactly equal to the corresponding data members of *that*. Returns false otherwise.

Parameters

that The **ccEdgeletChainFilterLength** object to compare with this object.

Public Member Functions

minChainLength

```
c_Int32 minChainLength() const;
```

Returns the minimum chain length.

filter

```
virtual void filter(
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const ccIndexChainList &inputChains,
  ccIndexChainList &filteredChains) const;
```

An override.

Performs a filtering operation on the given chains based on the chain length. A new **ccIndexChainList** will be formed by eliminating the chains from *inputChains* whose length is less than **minChainLength()**.

Parameters

inputEdgelets The vector of edgelets referenced by *inputChains*.

inputChains The input chains to be filtered.

filteredChains The filtered chains created when you execute this function.

ccEdgeletChainFilterMagnitudeHysteresis

```
#include <ch_cvl/edgefilt.h>

class ccEdgeletChainFilterMagnitudeHysteresis:
                                        public ccEdgeletChainFilter;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This is a filter class derived from **ccEdgeletChainFilter**. It provides functionality for filtering **ccEdgelet** chains based on the hysteresis magnitude of the **ccEdgelet**s and their chains. This filter can also be used as a simple **ccEdgelet** magnitude filter by using the same high and low threshold values.

Please see the **ccEdgeletChainFilter** base class reference page for additional information about using this class.

Constructors/Destructors

ccEdgeletChainFilterMagnitudeHysteresis

```
ccEdgeletChainFilterMagnitudeHysteresis(
  c_Int32 lowMagThresh = 1,
  c_Int32 highMagThresh = 1);
```

Constructs an object with the given low and high magnitude threshold values.

Parameters

lowMagThresh The low hysteresis magnitude threshold.

highMagThresh The high hysteresis magnitude threshold.

Throws

ccEdgeletChainFilterDefs::BadParams
If *highMagThresh* < *lowMagThresh*,
or if *lowMagThresh* < 1,
or if *highMagThresh* < 1.

Operators

operator==

```
bool operator== (const
  ccEdgeletChainFilterMagnitudeHysteresis &that) const;
```

Returns true if the internal data members of this object are exactly equal to the corresponding data members of *that*. Returns false otherwise.

Parameters

that — The **ccEdgeletChainFilterMagnitudeHysteresis** object to compare with this object.

operator!=

```
bool operator!= (const
  ccEdgeletChainFilterMagnitudeHysteresis &that) const;
```

Returns true if the internal data members of this object are not exactly equal to the corresponding data members of *that*. Returns false otherwise.

Parameters

that — The **ccEdgeletChainFilterMagnitudeHysteresis** object to compare with this object.

Public Member Functions

lowMagThresh

```
c_Int32 lowMagThresh() const;
```

Returns the low hysteresis magnitude threshold.

highMagThresh

```
c_Int32 highMagThresh() const;
```

Returns the high hysteresis magnitude threshold.

filter

```
virtual void filter(
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const ccIndexChainList &inputChains,
  ccIndexChainList &filteredChains) const;
```

An override.

This function performs a hysteresis threshold filtering operation on the given edgelets and chains. The following rules apply:

1. Edgelets with a magnitude less than **lowMagThresh()** are filtered out.
2. Edgelets with a magnitude greater than, or equal to **lowMagThresh()** are retained if they are in a chain that includes at least one edgelet with a magnitude greater than, or equal to **highMagThresh()**.

   Note that chains without at least one edgelet magnitude greater than, or equal to **highMagThresh()** are completely eliminated.
3. A new **ccIndexChainList** is created using the surviving edgelets by eliminating chains, trimming chains by eliminating edgelets at either end, or by breaking chains into multiple chains. If a chain is broken by eliminating an interior edgelet, the two resulting chains cannot be joined.

Parameters

inputEdgelets	The input edgelets associated with *inputChains*.
inputChains	The input chains to be filtered.
filteredChains	The filtered chains created when you execute this function.

ccEdgeletChainFilterMagnitudeHysteresis

ccEdgeletChainFilterShape

```
#include <ch_cvl/edgefilt.h>

class ccEdgeletChainFilterShape: public ccEdgeletChainFilter
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This is a filter class derived from **ccEdgeletChainFilter**. This filter provides functionality for filtering **ccEdgelet**s that are aligned with, and are within a specified distance from, any of the contours described by a supplied shape. Note that the shape model information (implicit or explicit) is used when doing this filtering operation. Weights are not used, but if you specify that the polarity should be used, the gradient direction of the edge must match the polarity of the specified shape for it to be filtered. One application of this filtering tool is to remove edge chains that are too close to a border of a region that is to be inspected. This avoids picking up normal variations near the border as defects.

Please see the **ccEdgeletChainFilter** base class reference page for additional information about using this class.

Constructors/Destructors

ccEdgeletChainFilterShape

```
ccEdgeletChainFilterShape(
  const ccShape &shape = cc2Point(),
  double distanceTol = HUGE_VAL,
  const ccRadian &angleTol = ccRadian(ckPI),
  bool ignorePolarity = true,
  const ccShape::ccSampleParams &samplingParams =
            ccShape::ccSampleParams(0.0, 0.1, ckMaxInt32,
                                                true, false));
```

Constructs a filter object using the given shape, distance and angle tolerance values, an edgelet polarity ignore flag, and sampling parameters.

Parameters

shape	The shape to use for filtering.
distanceTol	Edgelets that are not within this distance tolerance to a shape edge will be filtered out.

angleTol Edgelet tangent angles not within this angular tolerance to the nearest shape edge tangent angle are filtered out.

ignorePolarity When set to true, edge polarity is ignored. When set false, the edge polarity must be the same as the edgelet polarity or the edgelet is filtered out.

samplingParams The shape sampling parameters.

Throws

ccEdgeletChainFilterDefs::BadParams
If *distanceTol* < 0,
or if *angleTol* < **ccRadian(0.0)**,
or if *ignorePolarity* is false and the effective ignore polarity flag for any portion of *shape* is true.

Notes

Please see the **ccShapeModel** class for information on how to specify the shape polarity.

Operators

operator==

```
bool operator== (
  const ccEdgeletChainFilterShape &that) const;
```

Returns true if the internal data members of this object are exactly equal to the corresponding data members of *that*. Returns false otherwise.

Parameters

that The **ccEdgeletChainFilterShape** object to compare with this object.

operator!=

```
bool operator!= (
  const ccEdgeletChainFilterShape &that) const;
```

Returns true if the internal data members of this object are not exactly equal to the corresponding data members of *that*. Returns false otherwise.

Parameters

that The **ccEdgeletChainFilterShape** object to compare with this object.

Public Member Functions

shape

```
ccShapePtrh_const shape() const;
```

Returns the shape object specified in the constructor.

distanceTol

```
double distanceTol() const;
```

Returns the distance tolerance specified in the constructor.

angleTol

```
ccRadian angleTol() const;
```

Returns the angle tolerance specified in the constructor.

ignorePolarity

```
bool ignorePolarity() const;
```

Returns the ignore polarity flag specified in the constructor.

samplingParams

```
const ccShape::ccSampleParams &samplingParams() const;
```

Returns the sampling parameters specified in the constructor.

filter

```
virtual void filter(
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const ccIndexChainList &inputChains,
  ccIndexChainList &filteredChains) const;
```

An override.

This function is an override. It performs a filtering operation on the input edgelets using the shape, distance tolerance, and angle tolerance provided in the constructor. The filter analyzes the input edgelets and edgelet chains.

First, the shape is sampled by calling the static member function **ccShapeModel::sampleWithPolarity()** and passing it the *samplingParams* provided in the **ccEdgeletChainFilterShape** constructor. For each input edgelet, the sampled points within a distance of <= *distanceTol* to the edgelet are found. If there are no such points, the edgelet is not filtered out and it survives. If there are such points, the angular differences between the tangents of these points and the edgelet tangent angle is measured. If any of these angular differences is <= angleTol, the edgelet is filtered out. Otherwise, the edgelet survives.

The angular differences are taken in the ccRadian(0.0) to ccRadian(ckPI) range if *ignorePolarity* is true. Otherwise, the angular differences will be taken in the ccRadian(0.0) to ccRadian(ck2PI) range to take the shape and edgelet polarity into

consideration. If angleTol is >= ccRadian(ckPI), filtering based on angle tolerance will not be performed by turning off the *computeTangents* flag of samplingParams. No filtering will be done for shapes if there are no tangent angles produced by sampling and angleTol is not >= ccRadian(ckPI). A new ccIndexChainList will be formed using the surviving edgelets by eliminating, trimming or breaking the input chains.

Parameters

inputEdgelets	The edgelets associated with *inputChains*.
inputChains	The chains to be filtered.
filteredChains	The filtered result chains.

Throws

Any errors thrown by **ccShapeModel::sampleWithPolarity()** will be caught here.

ccEdgeletDefs

```
#include <ch_cvl/edge.h>

class ccEdgeletDefs;
```

A name space that holds enumerations and constants used with the Edge tool.

Enumerations

EdgeType

```
enum EdgeType;
```

This enumeration defines the types of edgelet representations that may be used to store a collection of edgelets within a **ccEdgletSet**.

Value	Meaning
eEdges	A vector of **ccEdgelet** objects that use 16-bit integers to store the whole-pixel location of the edge.
eEdges2	A vector of **ccEdgelet2** objects that use 32-bit integers to store the whole-pixel location of the edge.
eEdgesAndOffsets	One or more blocks of memory, where each block includes a 32-bit image offset specifier as well as a vector of **ccEdgelet** objects. This representation is more efficient and requires less memory when using very large images.

EdgeTypeRequest

```
enum EdgeTypeRequest;
```

This enumeration defines the types of edgelet representations that you may request that the tool use when storing edges within a **ccEdgletSet**. The actual representation used is returned by the tool.

Value	Meaning
eRequestContiguous	Request that edgelets be stored in a contiguous block of **ccEdgelet** objects. The resulting storage type may be *eEdges* or *eEdges2*.
eRequestEdges2	Request that edgelets be stored in a contiguous block of **ccEdgelet2** objects. The resulting storage type will be *eEdges2*.
eRequestEdgesAndOffsets	Request that edgelets be multiple blocks of **ccEdgelet** objects with an offset specifier for each block. The resulting storage type may be *eEdgesAndOffsets* or *eEdges* (if all the edges have whole-pixel image coordinates within the range -32768 to +32767.

ccEdgeletIterator

```
#include <ch_cvl/edge.h>

class ccEdgeletIterator: public ccEdgeletIterator_const;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	None

This class represents a single edgelet in a **ccEdgeletSet**. Unlike **ccEdgelet** and **ccEdgelet2**, this class can represent an edgelet regardless of the underlying storage method used by the **ccEdgeletSet**.

ccEdgeletIterator provides the same public accessor methods as **ccEdgelet** and **ccEdgelet2**. It also implements the standard STL iterator methods and operators for accessing edgelets within the **ccEdgeletSet**.

Note This class is derived from **ccEdgeletIterator_const**, which provides the most commonly-used methods for accessing edgelets. For more information, see the description of that class on page 1271.

Constructors/Destructors

ccEdgeletIterator

```
ccEdgeletIterator();

ccEdgeletIterator(c_Int32 edgeletIndex,
   ccEdgeletSet* set);
```

- `ccEdgeletIterator();`

  Constructs a null **ccEdgeletIterator**. The constructed object cannot be used.

- ```
  ccEdgeletIterator(c_Int32 edgeletIndex,
     ccEdgeletSet* set);
  ```

  Constructs a **ccEdgeletIterator** that points to the specified edgelet within the supplied **ccEdgeletSet**.
  ```

Parameters

edgeletIndex The index of the edgelet.

set The **ccEdgeletSet** to iterate upon.

Notes

The default copy constructor, assignment operator, and destructor are used.

The copy constructor and assignment operator are shallow, and the iterator will have the same edgelet set lifetime constraints as the iterator being copied or assigned from.

Operators

operator-

```
c_Int32 operator-(const ccEdgeletIterator& rhs) const;
```

Returns the difference between two iterators. Both iterators must refer to the same **ccEdgeletSet**.

Parameters

rhs The iterator to subtract from this one.

Standard STL Iterator Operators

The operators in this section implement the standard iterator increment, decrement, addition, subtraction, and comparison operators.

Notes

The comparison operators require that the *rhs* argument be an iterator that refers to the same **ccEdgeletSet**.

operator++

```
ccEdgeletIterator& operator++();
ccEdgeletIterator operator++(int);
```

operator--

```
ccEdgeletIterator& operator--();
ccEdgeletIterator operator--(int);
```

operator+=

```
ccEdgeletIterator& operator+=(c_Int32 offset);
```

operator+

```
ccEdgeletIterator operator+(c_Int32 offset) const;
```

operator-

```
ccEdgeletIterator operator-(c_Int32 offset) const;
```

operator-=

```
ccEdgeletIterator& operator-=(c_Int32 offset);
```

Public Member Functions

magnitude

```
void magnitude(c_UInt16 m);
```

Sets the gradient magnitude for this edgelet.

Parameters

m	The magnitude to set.

angle

```
void angle(const ccAngle8& a);
```

Sets the positive gradient angle for this edgelet.

Parameters

a	The angle to set.

ccEdgeletIterator

ccEdgeletIterator_const

```
#include <ch_cvl/edge.h>

class ccEdgeletIterator_const;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	None

This class represents a single edgelet in a **ccEdgeletSet**. Unlike **ccEdgelet** and **ccEdgelet2**, this class can represent an edgelet regardless of the underlying storage method used by the **ccEdgeletSet**.

ccEdgeletIterator_const provides the same public accessor methods as **ccEdgelet** and **ccEdgelet2**. It also implements the standard STL iterator methods and operators for accessing edgelets within the **ccEdgeletSet**.

Note This class provides const access to the **ccEdgeletSet**; for non-const access, use **ccEdgeletIterator**.

Constructors/Destructors

ccEdgeletIterator_const

```
ccEdgeletIterator_const();

ccEdgeletIterator_const(c_Int32 edgeletIndex,
   const ccEdgeletSet* set);
```

- `ccEdgeletIterator_const();`

  Constructs a null **ccEdgeletIterator_const**. The constructed object cannot be used.

- `ccEdgeletIterator_const(c_Int32 edgeletIndex, const ccEdgeletSet* set);`

  Constructs a **ccEdgeletIterator_const** that points to the specified edgelet within the supplied **ccEdgeletSet**.

  Parameters

  edgeletIndex The index of the edgelet.

set The **ccEdgeletSet** to iterate upon.

Notes

The default copy constructor, assignment operator, and destructor are used.

The copy constructor and assignment operator are shallow, and the iterator will have the same edgelet set lifetime constraints as the iterator being copied or assigned from.

Public Member Functions

isEnd `bool isEnd() const;`

Returns whether this iterator is done forward iterating, i.e. whether it has reached the end of the edgelet set.

Notes

Using this function may be significantly more efficient than using the more STL-like syntax

```
if (iter != end) ...
```

isRend `bool isRend() const;`

Returns whether this iterator is done reverse iterating, that is whether it has reached the reverse end of the edgelet set (one edgelet before the beginning).

Notes

Using this function may be significantly more efficient that using the more STL like syntax

```
if (iter != eend) ...
```

isDone `bool isDone() const;`

Returns whether this iterator is done iterating, that is whether it has reached either the forward or reverse end of the edgelet set.

index `c_Int32 index() const;`

Returns the index of the edgelet within the **ccEdgeletSet**. The returned value will be in the range 0 through **ccEdgeletSet::nEdges()** -1.

Notes

A newly constructed iterator returned by **ccEdgeletSet::begin()** will have an **index()** of 0.

setIndex `void setIndex(c_Int32 edgeletIndex);`

Set this iterator to refer to the specified edgelet index.

Parameters

edgeletIndex The index. *edgeletIndex* must be in the range 0 through **ccEdgeletSet::nEdges()** -1.

position `cc2Vect position() const;`

Returns the sub-pixel location of this edgelet.

posQuickX `double posQuickX(void) const;`

Returns the whole-pixel x-axis location of this edgelet with the lowest possible overhead, assuming that the edge location has been determined.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

posQuickY `double posQuickY(void) const;`

Returns the whole-pixel y-axis location of this edgelet with the lowest possible overhead, assuming that the edge location has been determined.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

posQuickR `double posQuickR(void) const;`

Returns the sub-pixel offset of this edgelet with the lowest possible overhead, assuming that the edge location has been determined.

This is the distance, in image coordinate system units, from the pixel center to the sub-pixel edge, measured along a line drawn through the pixel center in the direction of the edge.

Notes

If the sub-pixel peak location has not been determined by a previous call to **ccEdgeletSet::doEdgeInterp()**, then this function will not be particularly efficient.

magnitude

```
c_UInt16 magnitude() const;
```

Returns the magnitude of this edgelet.

angle

```
ccAngle8 angle() const;
```

Returns the angle of this edgelet, relative to the image coordinate system x-axis.

gradient

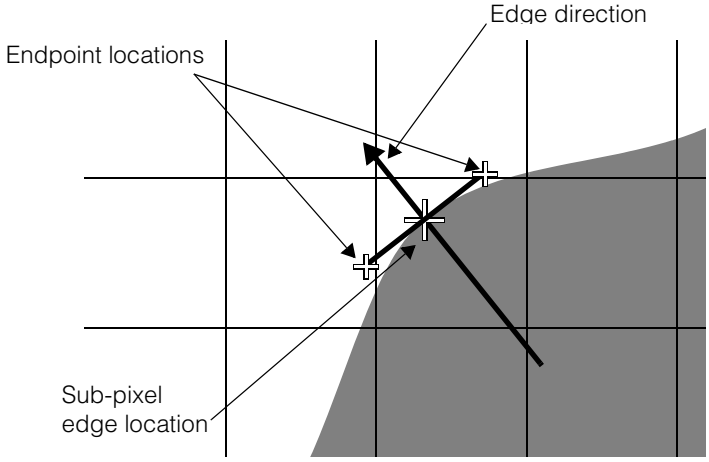
```
cc2Vect gradient() const;
```

Returns a unit vector in positive gradient direction (dark-to-light) that describes this edgelet.

endpoints

```
void endpoints(cc2Vect& left, cc2Vect& right, bool
useSubpixel = true) const;
```

Returns the location of a one-pixel-long line perpendicular to this edgelet's edge direction and drawn through the sub-pixel edge location, as shown in the following figure:



Parameters

left	A reference to the left endpoint of the line
right	A reference to the right endpoint of the line
useSubpixel	If true, locate the line with sub-pixel accuracy; if false, locate the line through the pixel center.

Notes

This function is intended to support the graphic display of edgelets.

Operators

operator-

```
c_Int32 operator-(
   const ccEdgeletIterator_const& rhs) const;
```

Returns the difference between two iterators. Both iterators must refer to the same **ccEdgeletSet**.

Parameters

rhs	The iterator to subtract from this one.

Standard STL Iterator Operators

The operators in this section implement the standard iterator increment, decrement, addition, subtraction, and comparison operators.

Notes

The comparison operators require that the *rhs* argument be an iterator that refers to the same **ccEdgeletSet**.

operator++

```
ccEdgeletIterator_const& operator++();
ccEdgeletIterator_const operator++(int);
```

operator--

```
ccEdgeletIterator_const& operator--();
ccEdgeletIterator_const operator--(int);
```

operator+=

```
ccEdgeletIterator_const& operator+=(c_Int32 offset);
```

operator+

```
ccEdgeletIterator_const operator+(c_Int32 offset) const;
```

operator-=

```
ccEdgeletIterator_const& operator-=(c_Int32 offset);
```

operator-

```
ccEdgeletIterator_const operator-(c_Int32 offset) const;
```

operator==

```
bool operator==(const ccEdgeletIterator_const& rhs) const;
```

operator!=

```
bool operator!=(const ccEdgeletIterator_const& rhs) const;
```

operator<

```
bool operator<(const ccEdgeletIterator_const& rhs) const;
```

operator>

```
bool operator>(const ccEdgeletIterator_const& rhs) const;
```

operator<=

```
bool operator<=(const ccEdgeletIterator_const& rhs) const;
```

operator>=

```
bool operator>=(const ccEdgeletIterator_const& rhs) const;
```

ccEdgeletParams

```
#include <ch_cvl/edge.h>

class ccEdgeletParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the parameters you use to control the generation of edge angle images, edge magnitude images, and **ccEdgeletSet**s.

Constructors/Destructors

ccEdgeletParams

```
ccEdgeletParams(double magScale = 1.0,
  double magThresh = 20.0, ccEdgeletDefs::EdgeTypeRequest
  edgeTypeRequest = ccEdgeletDefs::kDefaultRequest);
```

Constructs a **ccEdgeletParams** using the supplied parameters

Parameters

magScale The magnitude scaling factor. The computed edge magnitude for each pixel is multiplied by *magScale* before being placed in the edge magnitude image.

magThresh The magnitude threshold. **ccEdgelet**s are only constructed for edges with a magnitude greater than *magThresh* (after having been multiplied by *magScale*).

edgeTypeRequest
The requested storage method for edgelets. The supplied value must be one of the following:

ccEdgeletDefs::eRequestContiguous
ccEdgeletDefs::eRequestEdges2
ccEdgeletDefs::eRequestEdgesAndOffsets

Throws

ccEdgeletDefs::BadParams
magScale or *magThresh* is less than 0 or *edgeTypeRequest* is not a member of **ccEdgeletDefs::EdgeTypeRequest**.

Public Member Functions

magScale

```
double magScale() const;
void magScale(double scale);
```

- `double magScale() const;`

  Returns the magnitude scaling factor of this **ccEdgeletParams**.

- `void magScale(double scale);`

  Sets this **ccEdgeletParams**'s magnitude scaling factor. The edge magnitude computed for each pixel in the input image is multiplied by this factor before being placed in the edge magnitude image.

  Parameters

  scale The magnitude scaling factor to set

  Notes

  A magnitude scaling factor of 0.89 guarantees a maximum magnitude equal to the maximum pixel value.

  For a 16-bit input image and a 8-bit output magnitude image, *scale* should be divided by 256. For an 8-bit input image and a 16-bit output magnitude image, *scale* must be greater than 1.0 in order to capture the full range of edge magnitudes in the input image.

  On some platforms, the magnitude may silently overflow the maximum magnitude image pixel value; on others, it may be clipped to the maximum pixel value.

magThresh

```
double magThresh() const;
void magThresh(double thresh);
```

- `double magThresh() const;`

  Returns the magnitude threshold value for this **ccEdgeletSet**.

- `void magThresh(double thresh);`

  Sets the magnitude threshold value for this **ccEdgeletSet**, expressed as the change in pixel values across the edge. Only edges with magnitudes greater than *thresh* will have **ccEdgelet**s created for them.

Parameters

thresh The threshold to set

Notes

This value does not affect the magnitude or angle images.

edgeTypeRequest

```
ccEdgeletDefs::EdgeTypeRequest edgeTypeRequest() const;

void edgeTypeRequest(ccEdgeletDefs::EdgeTypeRequest e);
```

- `ccEdgeletDefs::EdgeTypeRequest edgeTypeRequest() const;`

  Returns the storage form specified for this **ccEdgeletParams** object. The returned value is one of the following

  ccEdgeletDefs::eRequestContiguous
  ccEdgeletDefs::eRequestEdges2
  ccEdgeletDefs::eRequestEdgesAndOffsets

- `void edgeTypeRequest(ccEdgeletDefs::EdgeTypeRequest e);`

  Sets the requested storage method for edgelets. The supported edge types are:

  - If the requested value is *ccEdgeletDefs::eRequestContiguous*, the edgelets will be stored in a single contiguous block. If no edgelet has a whole-pixel location with an image X- or Y-coordinate outside the range -32768 through +32767, the edgelets are stored in a vector of **ccEdgelet** and accessed through **ccEdgeletSet::edges()**. If any edgelet has a whole pixel location with an image X- or Y-coordinate that is outside the range outside the range -32768 through +32767, the edgelets are stored in a vector of **ccEdgelet2** and accessed through **ccEdgeletSet::edges2()**. In both cases, the edges can be accessed using the **ccEdgeletIterator** or **ccEdgeletIterator_const** classes obtained by calling **ccEdgeletSet::begin()**.
  - If the requested value is ccEdgeletDefs::eRequestEdges2, the edgelets are stored in a vector of **ccEdgelet2** and accessed through **ccEdgeletSet::edges2()** or by using the **ccEdgeletIterator** or **ccEdgeletIterator_const** classes obtained by calling **ccEdgeletSet::begin()**.
  - If the requested value is *ccEdgeletDefs::eRequestEdgesAndOffsets*, the edgelets are stored in one or more separate blocks of memory, where each block contains an array of **ccEdgelet** objects and a pair of 32-bit offset specifiers. Edgelets are accessed using the **ccEdgeletIterator** or **ccEdgeletIterator_const** classes. You obtain one of these objects by calling **ccEdgeletSet::begin()**.

Multiple blocks are only used when at least one edgelet has a whole pixel location with an image X- or Y-coordinate that is outside the range outside the range -32768 through +32767. If only a single block is created, you can use both **ccEdgeletSet::edges()** and the **ccEdgeletIterator()** methods for accessing the set's edgelets.

ccEdgeletSet::edgeTypes() indicates the actual edgelet storage method for a given **ccEdgeletSet** (which may not be the same as the requested method). While the **ccEdgeletSet::edges()** and **ccEdgeletSet::edges2()** edgelet accessors may only be used if the corresponding storage type was used, The **ccEdgeletIterator()** and **ccEdgeletIterator_const()** methods may always be used.

The default requested storage method is *ccEdgeletDefs::eRequestContiguous*.

Parameters

e — The requested storage type. *e* must be one of the following values:

ccEdgletDefs::eRequestContiguous
ccEdgletDefs::eRequestEdges2
ccEdgletDefs::eRequestEdgesAndOffsets

Notes

In general, *eRequestEdgesAndOffsets* is the most efficient form in terms of speed and memory usage of edgelet generation, as the other forms may require copying the edgelets. However, if the edgelets will need to be accessed in random order (such as when processing chains with a **ccIndexChainList**), the overall processing may be more efficient if the edgelets are copied into contiguous storage, since random access using **ccEdgeletIterator()** into discontiguous storage is less efficient that random access into contiguous storage.

ccEdgeletSet

```
#include <ch_cvl/edge.h>

class ccEdgeletSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains all the edgelets located in the input image.

The edgelets in this object may be stored in one of three ways, depending on the storage method that you requested with **ccEdgeletParams::edgeTypeRequest** and on whether or not any edgelet has a whole pixel location with an image X- or Y-coordinate that is outside the range -32768 through +32767.

ccEdgeletSet provides a single method, **ccEdgeletSet::begin()** that allows you to use a **ccEdgeletIterator** or **ccEdgeletIterator_const** object to access all of the **ccEdgeletSet**'s edgelets, regardless of how they are stored.

Cognex recommends the use of **ccEdgeletSet::begin()** rather than the storage-type specific **edges()** and **edges2()** accessors.

For more information, see the function **ccEdgeletParams::edgeTypeRequest()** on page 1279 and the function **ccEdgeletSet::edgeTypes()** on page 1287.

Constructors/Destructors

ccEdgeletSet

```
ccEdgeletSet();

ccEdgeletSet(const cmStd vector<ccEdgelet>& set,
  double magScale = 1.0);
```

- `ccEdgeletSet();`

  Constructs an unbound **ccEdgeletSet**.

- `ccEdgeletSet(const cmStd vector<ccEdgelet>& edgelets,
  double magScale = 1.0);`

  Constructs a **ccEdgeletSet** from the supplied vector of **ccEdgelet**s.

Parameters

edgelets	The **ccEdgelet**s for this **ccEdgeletSet**
magScale	The edge magnitude scaling factor that was used to construct the **ccEdgelet**s in *edgelets*

Public Member Functions

setUnbound

```
void setUnbound();
```

Deletes all of the **ccEdgelet**s in this **ccEdgeletSet** but does not affect the **cc2Xform** or magnitude scaling factor associated with this **ccEdgeletSet**.

isBound

```
bool isBound() const;
```

Returns true if this **ccEdgeletSet** contains at least one **ccEdgelet**.

copyXforms

```
void copyXforms(const ccEdgeletSet& other);
```

Efficiently sets this **ccEdgeletSet** object's **imageFromClientXformBase** and **ClientFromImageXformBase** transforms to be the same as the supplied **ccEdgeletSet** object's transforms.

Parameters

other	The object containing the transforms to set.

Notes

This function is the most efficient way (in both speed and memory) to make two **ccEdgeletSet** objects use the same transform; it is much more efficient than calling the two objects' setters and getters. Also, this method avoids any potential loss of accuracy caused by inverting a transform twice.

The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

clientFromImageXformBase

```
const cc2XformBasePtrh_const&
   clientFromImageXformBase() const;

void clientFromImageXformBase(
   const cc2XformBasePtrh_const& xform, bool copy = true);

void clientFromImageXformBase(const cc2XformBase& xform);
```

- ```
  const cc2XformBasePtrh_const&
     clientFromImageXformBase() const;
  ```

  Returns a **cc2XformBase** that describes the transformation between the coordinate system in which the edgelet positions are given and the client coordinate system of the input image used to create this **ccEdgeletSet**.

  Notes

  The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

- ```
  void clientFromImageXformBase(
     const cc2XformBasePtrh_const& xform, bool copy = true);
  ```

  Sets the **cc2XformBase** that describes the transformation between the coordinate system in which the edgelet positions are given and the client coordinate system of the input image used to create this **ccEdgeletSet**.

  Parameters

xform	The transformation.
copy	If true, a deep copy is made of *xform*; if false, a reference is stored. If *copy* is false, the supplied transform must not be modified during the lifetime of this **ccEdgeletSet**.

  Notes

  The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

- ```
  void clientFromImageXformBase(const cc2XformBase& xform);
  ```

  Sets the **cc2XformBase** that describes the transformation between the coordinate system in which the edgelet positions are given and the client coordinate system of the input image used to create this **ccEdgeletSet**.
  ```

Parameters

xform The transformation.

Notes

This overload makes a deep copy of the supplied transformation.

The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

imageFromClientXformBase

```
const cc2XformBasePtrh_const&
   imageFromClientXformBase() const;

void imageFromClientXformBase(
   const cc2XformBasePtrh_const& xform, bool copy = true);

void imageFromClientXformBase(const cc2XformBase& xform);
```

- ```
  const cc2XformBasePtrh_const&
     imageFromClientXformBase() const;
  ```

  Returns a **cc2XformBase** that describes the transformation between the client coordinate system of the input image used to create this **ccEdgeletSet** and the coordinate system in which the edgelet positions are given.

  Notes

  The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

- ```
  void imageFromClientXformBase(
     const cc2XformBasePtrh_const& xform, bool copy = true);
  ```

  Sets the **cc2XformBase** that describes the transformation between the client coordinate system of the input image used to create this **ccEdgeletSet** and the coordinate system in which the edgelet positions are given.

  Parameters

  xform The transformation.

  copy If true, a deep copy is made of *xform*; if false, a reference is stored. If *copy* is false, the supplied transform must not be modified during the lifetime of this **ccEdgeletSet**.

Notes

The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

- `void imageFromClientXformBase(const cc2XformBase& xform);`

Sets the **cc2XformBase** that describes the transformation between the client coordinate system of the input image used to create this **ccEdgeletSet** and the coordinate system in which the edgelet positions are given.

Parameters

xform The transformation.

Notes

This overload makes a deep copy of the supplied transformation.

The **imageFromClientXformBase** and **ClientFromImageXformBase** methods both refer to the same underlying object (as do the deprecated **imageFromClientXform** and **ClientFromImageXform** methods).

nEdges `c_Int32 nEdges() const;`

Returns the number of edgelets in this **ccEdgeletSet**.

magScale `double magScale() const;`

Returns magnitude scaling factor used to compute the magnitudes associated with the edgelets in this **ccEdgeletSet**.

doEdgeInterp `void doEdgeInterp();`

Forces the computation of sub-pixel offsets for each edgelet in this **ccEdgeletSet**.

begin

```
ccEdgeletIterator begin();

ccEdgeletIterator_const begin() const;
```

- `ccEdgeletIterator begin();`

  Gets a **ccEdgeletIterator** object that you can use to access the edgelets in this **ccEdgeletSet**. This is the recommended method for accessing edgelets, as it is valid regardless of the storage method used by this **ccEdgeletSet**.

  You can use several methods to iterate using a **ccEdgeletIterator** object:

  - You can use the **ccEdgeletIterator::isEnd()**:

    ```
    for (ccEdgeletIterator_const iter = edgeletSet.begin();
         !iter.isEnd());
         ++iter)
         {
         cc2Vect pos = iter.position();
         }
    ```

  - You can use **ccEdgeletSet::nEdges()** to determine the extent of the edgelet set:

    ```
    c_Int32 nEdges = edgeletSet.nEdges();
    ccEdgeletIterator_const iter = edgeletSet.begin();
    for (c_Int32 i = 0; i < nEdges; ++i, ++iter)
         {
         cc2Vect pos = iter.position();
         }
    ```

  - You can also use the iterator to obtain random access to an edgelet, instead of the sequential access shown above, by using the **ccEdgeletIterator::setIndex()** or the slightly less efficient **ccEdgeletIterator::operator+** overload:

    ```
    ccEdgeletIterator_const iter = edgeletSet.begin();
    const ccIndexChain& chain = chainList.chain[chainIndex];
    const c_Int32* start = chain.start();
    c_Int32 n = chain.length();
    for (c_Int32 i = 0; i < n; i++)
       {
       c_Int32 edgeletIndex = start[i];
       iter.setIndex(edgeletIndex);
       cc2Vect pos = iter.position();
       }
    ```

- `ccEdgeletIterator_const begin() const;`

  This method provides const access to the edgelets. For more information see the class **ccEdgeletIterator_const** on page 1271.

edgeTypes `c_UInt32 edgeTypes() const;`

Returns the storage methods used in this **ccEdgeletSet** to store edgelets. The returned value is formed by ORing together one or more of the following values:

ccEdgeletDefs::eEdges
ccEdgeletDefs::eEdges2
ccEdgeletDefs::eEdgesAndOffsets

Notes

An empty edgelet set will return *ccEdgeletDefs::eEdges* | *ccEdgeletDefs::eEdges2*. All other cases will return only a single bit.

If *ccEdgeLetDefs::eRequestEdgesAndOffsets* was specified for the requested edge type and all of the edgelets had whole-pixel locations with X- and Y-coordinates in the range -32768 through +32767, **edgeTypes** will return *ccEdgeletDefs::eEdges*.

hasEdges `bool hasEdges() const`

Returns true if this **ccEdgeletSet** stores edgelets as a vector of **ccEdgelet**, accessible by calling **ccEdgeletSet::edges()**.

This function returns true in two cases:

- *ccEdgeLetDefs::eRequestContiguous* was specified for the requested edge type and all of the edgelets had whole-pixel locations with X- and Y-coordinates in the range -32768 through +32767.
- *ccEdgeLetDefs::eRequestEdgesAndOffsets* was specified for the requested edge type and all of the edgelets had whole-pixel locations with X- and Y-coordinates in the range -32768 through +32767.

hasEdges2

```
bool hasEdges2() const;
```

Returns true if this **ccEdgeletSet** stores edgelets as a vector of **ccEdgelet2**, accessible by calling **ccEdgeletSet::edges2()**.

This function returns true in two cases:

- *ccEdgeLetDefs::eRequestContiguous* was specified for the requested edge type and at least one of the edgelets had whole-pixel locations with X- and Y-coordinates outside of the range -32768 through +32767.
- *ccEdgeLetDefs::eRequestEdges2* was specified for the requested edge type.

hasEdgesAndOffsets

```
bool hasEdgesAndOffsets() const;
```

Returns true if this **ccEdgeletSet** stores edgelets in two or more possibly non-contiguous blocks of memory along with 32-bit offsets for each block. If this function returns true, you must use the **ccEdgeletIterator** or **ccEdgletIterator_const** returned by **begin()** to access the edgelets in this set.

This function returns true in one case:

- *ccEdgeLetDefs::eRequestEdgesAndOffsets* was specified for the requested edge type and at least one of the edgelets had whole-pixel locations with X- and Y-coordinates outside of the range -32768 through +32767.

edges

```
const cmStd vector<ccEdgelet>& edges() const;

void edges(const cmStd vector<ccEdgelet>& edges);
```

- ```
  const cmStd vector<ccEdgelet>& edges() const;
  ```

  Returns a reference to a vector containing all of the **ccEdgelet**s in this **ccEdgeletSet**.

  Notes

  This function may only be called if **hasEdges()** returns true. By using the **begin()** function instead of this one, you can be assured that your code will work correctly regardless of how the edgelets in this **ccEdgeletSet** are stored.

- ```
  void edges(const cmStd vector<ccEdgelet>& edges);
  ```

  Replaces all the **ccEdgelet**s in this **ccEdgeletSet** with the supplied vector of **ccEdgelet**s.

Parameters

edges The **ccEdgelet**s to set

Notes

If *edges* contains no **ccEdgelet**s, the effect is equivalent to calling **setUnbound()**.

This function also sets the **edges2()** vector to an empty vector and clears any edgelets stored in multiple storage blocks.

---

edges2

```
const cmStd vector<ccEdgelet2>& edges2() const;
void edges2(const cmStd vector<ccEdgelet2>&);
```

- `const cmStd vector<ccEdgelet2>& edges2() const;`

Returns a reference to a vector containing all of the **ccEdgelet2**s in this **ccEdgeletSet**.

Notes

This function may only be called if **hasEdges2()** returns true. By using the **begin()** function instead of this one, you can be assured that your code will work correctly regardless of how the edgelets in this **ccEdgeletSet** are stored.

- `void edges2(const cmStd vector<ccEdgelet2>&);`

Replaces all the **ccEdgelet2**s in this **ccEdgeletSet** with the supplied vector of **ccEdgelet2**s.

Notes

This function also sets the **edges()** vector to an empty vector and clears any edgelets stored in multiple storage blocks.

boundingBox

```
ccPelRect boundingBox() const;
```

Returns the minimum enclosing rectangle of the whole pixel positions in this **ccEdgeletSet**. The rectangle is in image coordinates. If there are no **ccEdgelet**s in this **ccEdgeletSet**, then a null **ccPelRect** is returned.

centerOfMass

```
cc2Vect centerOfMass() const;
```

Returns the center of mass of the **ccEdgelet**s in this **ccEdgeletSet** in image coordinates.

Throws

ccMathError::Singular

This **ccEdgeletSet** contains no **ccEdgelet**s.

centerOfProjection

```
cc2Vect centerOfProjection() const;
```

Returns the point that minimizes the RMS distance to all the **ccEdgelet**s in this **ccEdgeletSet**, where **ccEdgelet**s are considered to have only 1 degree of freedom. (The point returned by **centerOfMass()** considers **ccEdgelet**s to have 2 degrees of freedom).

For many figures, the center of projection has a useful meaning. For example, for a corner of any size, the center of projection is at the corner point.

Throws

ccMathError::Singular
This **ccEdgeletSet** contains no **ccEdgelet**s or the center of projection is undefined.

Operators

operator=

```
ccEdgeletSet& operator=(const ccEdgeletSet& rhs);
```

Assignment operator.

Parameters

rhs The object to assign to this one.

Deprecated Members

clientFromImageXform

```
cc2Xform clientFromImageXform() const;

void clientFromImageXform(
  const cc2Xform& clientFromImage);
```

This function is deprecated. Use **clientFromImageXformBase()** instead.

- `cc2Xform clientFromImageXform() const;`

  Returns the **cc2Xform** that describes the transformation between the client coordinate system of the input image used to create this **ccEdgeletSet** and the coordinate system in which the **ccEdgelet** positions are given.

- ```
  void clientFromImageXform(
    const cc2Xform& clientFromImage);
  ```

  Sets the **cc2Xform** that describes the transformation between the client coordinate system of the input image used to create this **ccEdgeletSet** and the coordinate system in which the **ccEdgelet** positions are given.

  Parameters
  clientFromImage The transformation to set

imageFromClientXform

```
cc2Xform imageFromClientXform() const;

void imageFromClientXform(
  const cc2Xform& imageFromClient);
```

This function is deprecated. Use **imageFromClientXformBase()** instead.

- ```
  cc2Xform imageFromClientXform() const;
  ```

  Returns the **cc2Xform** that describes the transformation between the coordinate system in which the **ccEdgelet** positions are given and the client coordinate system of the input image used to create this **ccEdgeletSet**.

- ```
  void imageFromClientXform(
    const cc2Xform& imageFromClient);
  ```

  Sets the **cc2Xform** that describes the transformation between the coordinate system in which the **ccEdgelet** positions are given and the client coordinate system of the input image used to create this **ccEdgeletSet**.

  Parameters
  imageFromClient

  The transformation to set
  ```

ccEdgeletSet

ccEllipse

```
#include <ch_cvl/oldshape.h>

class ccEllipse;
```

This class is deprecated. Use the **ccEllipse2** class instead.

ccEllipse

ccEllipse2

```
#include <ch_cvl/ellipse.h>

class ccEllipse2 : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccEllipse2** class represents arbitrary 2D ellipses. The ellipses may be degenerate, that is, one or both of the radii may vanish. (The term *radii* refers to the half-lengths of the major and minor axes of an ellipse.) More precisely, **ccEllipse2** represents an arbitrary 2D affine transformation of the unit circle. This definition is more general than defining an ellipse simply as a locus of points. The affine transformation definition allows you to parameterize the ellipse by an angle in a well-defined way, and either in a right-handed or left-handed fashion.

Degrees of Freedom

When considered simply as a locus of points, an ellipse has five degrees of freedom (DOF): a center point, two radii, and an orientation. (The center point is a vector (x,y), so it has two DOF.)

An alternative view is to consider an ellipse to be an arbitrary affine transformation of the unit circle, that is, as the set of points (*x*, *y*) generated by the mapping:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} \cos\Phi \\ \sin\Phi \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

where *a*, *b*, *c*, *d*, *e*, and *f* are constant real values, and ϕ is an angle varying from 0 to 2π radians. The vector ($cos\phi$, $sin\phi$) parameterizes points on the unit circle in the standard way. The matrix multiplication and vector addition map these points to the ellipse.

The six DOF in the affine transformation do not contradict the fact that ellipses have but five DOF when viewed as loci of points. The paradox is resolved by noting that for any ellipse in the plane, viewed as a locus of points, there are an infinite number of mappings of the unit circle to that ellipse. For one particular point on the unit circle, such as the point (1, 0), one can arbitrarily choose to where that point is mapped on the ellipse.

Once this correspondence has been established, the affine transformation is completely determined, and all of the other points on the unit circle are mapped to the ellipse in a continuous fashion. The freedom to choose the mapped destination of the point (1, 0) accounts for the extra degree of freedom in the affine transformation representation. In **ccEllipse2**, this extra degree of freedom is called the *phase* of the ellipse.

To completely determine the affine transformation of the unit circle, you must also specify in which direction the mapped unit circle points proceed around the ellipse. This direction is called the *handedness* of the ellipse, and corresponds to the sign of the determinant of the 2 x 2 matrix of the affine transformation.

If an application treats ellipses purely as geometric objects, and never requires nor refers to a parameterization of points along the ellipse, then the phase and handedness of the ellipse are irrelevant and can be ignored. On the other hand, if you want to define arcs of the ellipse based on ranges of an angular parameter, or if you want to know the parameter value (not simply the 2D coordinates) of, for example, the closest point on the ellipse, then phase and handedness play a role.

Ellipse Parameterization

The **ccEllipse2** class can map an angle ϕ to a point on the ellipse. For example, if *X* is a **cc2XformLinear** object that maps points on the unit circle to points on the ellipse, then ϕ is mapped to the point **X.mapPoint(cc2Vect(1, ϕ))**.

In terms of the individual DOF, the mapping proceeds as follows:

1. $\phi' = \phi$ + *phase()*
2. if (!*isRightHanded()*)
   $\phi' = -\phi'$
3. *p = cc2Vect(1, ϕ')*
4. *p.x() *= radii().x()*;
   *p.y() *= radii().y()*
5. Rotate *p* by *angle()*
6. Translate *p* by *center()*

After the above sequence, *p* is the point on the ellipse corresponding to ϕ. The transformation from the ellipse to the unit circle, if it exists, is achieved by negating the sense of each of the above steps and performing them in reverse order.

DOF versus Transform Representations

The **ccEllipse2** class provides two representations of an ellipse: the DOF representation and the transform representation. The DOF representation includes a constructor plus setters and getters for all of the individual DOFs (center, radii, orientation, phase) as well as handedness. The transform representation includes a constructor, setter, and getter that take or return the full **cc2XformLinear** object that maps the unit circle to the ellipse.

Within a single representation, all of the values are exactly consistent. The DOF getters return the exact values passed to the corresponding DOF setters or the DOF constructor. Likewise, the transform getter returns the exact value passed to the transform setter or the transform constructor.

Mixing representations is legal, but has some consistency issues. When using a DOF constructor or setter, the transform representation is recomputed from the values of the DOF. Conversely, when using the transform constructor or setter, the entire DOF representation is recomputed from the transform. The decomposition of the transform into individual DOFs is almost unique, but not completely. **ccEllipse2** uses a heuristic that attempts to pick the *least surprising* DOF representation when the transform is modified: it minimizes the change in phase. This tends to prevent swapping of the radii, and to reflect changes in rotation of the transform in the angle DOF. For example, premultiplying the transform by a pure rotation by θ will adjust the angle by θ but leave the phase unchanged. There is still a possibility of unexpected changes in the individual DOF when modifying the transform.

Mapping Functions

Mapping of ellipses using **map()** is based on the transform representation. The transform of the mapped ellipse is the composition of the mapping transform with the transform of the original ellipse. This invariant holds for any **ccEllipse2** *e* and **cc2XformLinear** *X*:

```
e.map(X).ellFromUnitCirc() == X * e.ellFromUnitCirc();
```

where == denotes exact equality. The DOF representation of the mapped ellipse must be recomputed from the new transform. This representation is not completely unique, so the values of the individual DOF may be different than expected after a map.

Consider an ellipse that is mapped by a transform that produces a new ellipse, centered at the origin, with x-intercepts of ±3 and y-intercepts of ±2. The mapped ellipse can be viewed as an ellipse with an x-radius of 3, a y-radius of 2, and an orientation of 0; or it can be viewed as an ellipse with an x-radius of 2, a y-radius of 3, and an orientation of $\pi/2$ radians. There are other variations that involve adjusting the orientation and the phase in tandem. The representation you ultimately choose depends on the values of the individual DOFs in the original ellipse.

The process of mapping ellipses may also introduce slight imprecision in DOF values as a result of floating point error. For example, mapping an ellipse by a rigid transform should leave the radii unchanged. However, this process may cause the radii values to change slightly, as they are recomputed from the mapped transform. Because of floating point error, even mapping by the identity transform can cause slight changes in the DOF values since they are recomputed from the transform in the mapped ellipse, whereas the transform may have been computed from the DOF in the original.

Because of issues associated with mapping ellipses, the **ccEllipse2** also provides three auxiliary mapping functions: **translate()**, **rotate()**, and **scale()**. Unlike **map()**, which operates on the transform representation, these operations operate on the DOF representation. For instance, **rotate()** adjusts the orientation DOF and leaves the other DOF unchanged; the transform is then recomputed from the DOF. These DOF-based mapping functions have some advantages over the transform-based **map()**. For simple mappings, such as a pure rotation or scaling, they are simpler to call and also faster than **map()**. They can easily be composed, as in:

```
e.rotate(theta).translate(offset);
```

Finally, they are guaranteed to leave certain DOFs unchanged. For example, rotating an ellipse via **rotate()** produces a new ellipse with exactly the same radii, whereas calling **map()** with a pure rotation transform may perturb the radii as a result of floating point error. In the first case, the transform of the mapped ellipse is recomputed from the DOF, while in the second case, the DOFs are recomputed from the transform.

Constructors/Destructors

ccEllipse2

```
ccEllipse2();

ccEllipse2(const cc2Vect &radii,
  const ccRadian &angle = ccRadian(0.0),
  const cc2Vect &center = cc2Vect(0.0, 0.0),
  bool isRightHanded = true,
  const ccRadian &phase = ccRadian(0.0));

ccEllipse2(const cc2XformLinear &ellFromUnitCirc);

ccEllipse2(double a, double b, double c, double d,
  double e, double f);

explicit ccEllipse2(const ccEllipse &ell);
```

- `ccEllipse2();`

  Default constructor. Constructs a right-handed unit circle, centered at the origin, with zero angle and phase.

- ```
  ccEllipse2(const cc2Vect &radii,
    const ccRadian &angle = ccRadian(0.0),
    const cc2Vect &center = cc2Vect(0.0, 0.0),
    bool isRightHanded = true,
    const ccRadian &phase = ccRadian(0.0));
  ```

  Constructs an ellipse with given values for the individual degrees of freedom and handedness.
  ```

Parameters

radii	The radii.
angle	The angle in radians.
center	The center.
isRightHanded	Whether the ellipse is right-handed (default = true)
phase	The phase.

Notes

The *angle* and *phase* degrees of freedom are internally normalized to lie within the range $-\pi$ through $+\pi$.

Throws

ccShapesError::BadParams
Either component of *radii* is negative.

- `ccEllipse2(const cc2XformLinear &ellFromUnitCirc);`

Conversion constructor. Constructs an ellipse by mapping the unit circle by the given affine transform.

Parameters

ellFromUnitCirc	The affine transform.

Notes

If *ellFromUnitCirc* specifies a circular ellipse, the initial phase is zero.

- `ccEllipse2(double a, double b, double c, double d, double e, double f);`

Constructs an ellipse that satisfies the equation:

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

The x values of the radii of the constructed ellipse are greater than or equal to the y values, the ellipse has zero phase, and is right-handed.

Parameters

a	A constant coefficient.
b	A constant coefficient.
c	A constant coefficient.

d A constant coefficient.

e A constant coefficient.

f A constant coefficient.

Throws

ccShapesError::BadParams
The combination of coefficients does not produce a valid ellipse.

Notes

Not all combinations of coefficients specify ellipses. Some specify other conic sections (for example, hyperbolas), lines, empty sets, or the entire 2D plane. Only combinations that specify ellipses are valid.

- `explicit ccEllipse2(const ccEllipse &ell);`

Conversion constructor. Constructs a **ccEllipse2** from a deprecated **ccEllipse**. The constructed **ccEllipse2** will have zero phase. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

ell The old-style ellipse.

Notes

This constructor is provided to support legacy code that might need to convert an old-style **ccEllipse** to a new-style **ccEllipse2**. It is not recommended as a way of constructing a **ccEllipse2** in new code.

Operators

operator== `bool operator==(const ccEllipse2 &rhs) const;`

Returns true if and only if this ellipse equals *rhs*. This method tests for exact equality of all internal data members.

Parameters

rhs The other ellipse.

operator!= `bool operator!=(const ccEllipse2 &rhs) const;`

Returns true if and only if this ellipse is not equal to *rhs*.

Parameters

rhs The other ellipse.

Public Member Functions

radii

```
const cc2Vect &radii() const;
void radii(const cc2Vect &r);
```

- `const cc2Vect &radii() const;`

  Gets the radii of this ellipse.

- `void radii(const cc2Vect &r);`

  Sets the radii of this ellipse.

  Parameters

  r — The radii. Both components of the **cc2Vect** must be greater than or equal to zero.

  Throws

  ccShapesError::BadParams — Either component of *r* is negative.

  Notes

  The radii are the half-lengths of the axes of symmetry. Both values are non-negative. Either may be the larger of the two (that is, either the x-component or the y-component of the radii vector may represent the major axis of the ellipse.

center

```
const cc2Vect &center() const;
void center(const cc2Vect &c);
```

- `const cc2Vect &center() const;`

  Gets the center of this ellipse.

- `void center(const cc2Vect &c);`

  Sets the center point of this ellipse.

  Parameters

  c — The center.

angle

```
ccRadian angle() const;
void angle(const ccRadian &a);
```

- `ccRadian angle() const;`

  Gets the orientation of this ellipse.

- `void angle(const ccRadian &a);`

  Sets the orientation of this ellipse.

  Parameters

  a The orientation as an angle in radians.

  Notes

  The orientation is the angle that the x-axis of symmetry (that is, the axis corresponding to **radii.x()**) makes with the positive x-axis. In determining the mapping from the unit circle to the ellipse, this value has significance over the full range of 2π radians. However, in determining the locus of points on the ellipse, the value is only significant modulo π radians.

phase

```
ccRadian phase() const;
void phase(const ccRadian &p);
```

- `ccRadian phase() const;`

  Gets the phase of the ellipse.

- `void phase(const ccRadian &p);`

  Sets the phase of this ellipse.

  Parameters

  p The phase in radians.

  Notes

  With handedness, the phase establishes the precise parameterization (correspondence) between angles and points on the ellipse. If the ellipse is right-handed, then the angle -phase (or if it is left-handed, the angle +phase) maps to the point:

```
center() + cc2Vect(radius().x(), angle())
```

Phase has no effect in determining the locus of points that lie on the ellipse. It is the sixth degree of freedom, absent in the geometric representation of an ellipse but present in the affine transformation representation. Changing the phase does not change the locus of points that lie on the ellipse. Setting the phase to zero is often a convenient way to establish a simple parameterization of the ellipse, where the zero angle maps to a point on the x axis of symmetry.

isRightHanded

```
bool isRightHanded() const;
void isRightHanded(bool isRtHanded);
```

- `bool isRightHanded() const;`

Returns true if this ellipse is right-handed. See **ccShape::isRightHanded()** for more information.

- `void isRightHanded(bool isRtHanded);`

Sets the handedness of this ellipse.

Parameters

isRtHanded True sets this ellipse to right-handed, false to left-handed.

Notes

With phase, the handedness establishes the precise correspondence between angles and points on the ellipse. See **phase()** on page 1302 for details. In a right-handed ellipse, as the angle increases the mapped points travel around the ellipse in the sense of the x-axis rotating into the y-axis. In a left-handed ellipse, the direction is reversed. Changing the handedness does not change the locus of points that lie on the ellipse.

ellFromUnitCirc

```
const cc2XformLinear &ellFromUnitCirc() const;
void ellFromUnitCirc(const cc2XformLinear &X);
```

- `const cc2XformLinear &ellFromUnitCirc() const;`

Gets the full affine transformation from the unit circle to this ellipse.

- `void ellFromUnitCirc(const cc2XformLinear &X);`

  Sets the full affine transformation from the unit circle to this ellipse. This transformation includes the effects of phase and handedness. See the constructor **ccEllipse2(const cc2XformLinear &ellFromUnitCirc);** on page 1299 for details.

  Parameters

  X The affine transformation object that transforms a unit circle to this ellipse.

  Notes

  A singular transform corresponds to a degenerate ellipse, in which at least one of the two radii is zero.

map

```
ccEllipse2 map(const cc2XformLinear &X) const;
```

Returns this ellipse mapped by the transformation object *X*.

Parameters

X The transformation object.

Notes

This method recomputes the radii, center, angle, phase, and handedness values for the mapped ellipse. These values may differ for the mapped ellipse from those of the original ellipse in ways not expected. See *Mapping Functions* on page 1297 for details. The **translate()**, **rotate()**, and **scale()** methods offer more controlled mappings that are guaranteed to leave certain DOFs completely unchanged.

translate

```
ccEllipse2 translate(const cc2Vect &offset) const;
```

Returns this ellipse translated by the vector *offset*.

Parameters

offset The vector containing the offset value.

Notes

The *radii*, *angle*, *phase*, and *handedness* values of the translated ellipse are identical to those of the original ellipse. Only the *center* is translated by *offset*.

This method is functionally equivalent to copy constructing a new ellipse *e* from this one, and then executing the sequence:

```
e.center(e.center() + offset);
```

rotate

```
ccEllipse2 rotate(const ccRadian &theta) const;
```

Returns this ellipse rotated about its center by the angle *theta*.

Parameters

theta The angle.

Notes

The *radii*, *center*, *phase*, and *handedness* values of the rotated ellipse are identical to those of the original ellipse. Only the *angle* is adjusted by *theta*.

This method is functionally equivalent to copy constructing a new ellipse *e* from this one, and then executing the sequence:

```
e.angle(e.angle() + theta);
```

scale

```
ccEllipse2 scale(const cc2Vect &mag) const;
```

Returns this ellipse scaled by the vector *mag* about its center and along the directions of its axes of symmetry.

Parameters

mag The vector containing the magnification value.

Notes

The *center*, *angle*, *phase*, and *handedness* values of the translated ellipse are identical to those of the original ellipse. The **radii().x()** value is scaled by **mag().x()**, and the **radii().y()** value is scaled by **mag.y()**.

Notes

This method is functionally equivalent to copy constructing a new ellipse *e* from this one, and then executing the sequence:

```
e.radii(cc2Vect(e.radii().x() * mag.x(),
                e.radii().y() * mag.y()));
```

Throws

ccShapesError::BadParams
Either component of *mag* is negative.

point

```
cc2Vect point(const ccRadian &phi) const;
```

Returns the point at parameter value ϕ along the ellipse.

Notes

The *phase* and *handedness* of the ellipse both affect this value.

tangent

```
ccRadian tangent(const ccRadian &phi) const;
```

Returns the tangent direction at parameter value ϕ along the ellipse.

Notes

The *phase* and *handedness* of the ellipse both affect this value.

Throws

ccShapesError::NoTangent

hasTangent() is false for this ellipse. This occurs if and only if both radii are zero.

phi

```
ccRadian phi(const cc2Vect &p) const;
```

Returns the value of the parameter ϕ corresponding to a given point on this ellipse.

Parameters

p The point.

Notes

This method assumes *p* is a point on (or very near) the ellipse, such as a point computed by **nearestPoint()**. The result returned for other points is undefined.

coeffs

```
void coeffs(double &a, double &b, double &c, double &d,
  double &e, double &f) const;
```

Returns a set of coefficients defining an equation satisfied by all points that lie on this ellipse:

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

Parameters

a A constant coefficient.

b A constant coefficient.

c A constant coefficient.

d A constant coefficient.

e A constant coefficient.

f A constant coefficient.

Notes

Not all ellipses have such an algebraic representation. Specifically, there is no algebraic representation for an ellipse with one zero radius and one nonzero radius.

Throws

ccShapesError::DegenerateShape

This ellipse has exactly one zero radius. (A fully degenerate point ellipse with both radii equal to zero will not cause a throw.)

isDegenerate `bool isDegenerate() const;`

Returns true if and only if at least one of the radii of this ellipse is zero.

clone `virtual ccShapePtrh clone() const;`

Returns a pointer to a copy of this ellipse.

isOpenContour `virtual bool isOpenContour() const;`

Returns true if this shape is an open contour. For ellipses, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion `virtual bool isRegion() const;`

Returns true if this shape is a region. For ellipses, this function always returns true, including ellipses that are degenerate. See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

For ellipses, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For ellipses, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

For ellipses, this function returns true if either of the radii is nonzero. It returns false if both radii are zero. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For ellipses, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible `virtual bool isReversible() const;`

For ellipses, this function always returns true. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this ellipse. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this ellipse to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

sample

```
void sample(const ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this ellipse.

Parameters

params	Parameters object specifying details of how the sampling should be done.
result	Result object to which position and tangent chains are stored.

Notes

If **params.computeTangents()** is true, this function ignores ellipses for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if the given point is within this ellipse.

Parameters

p	The point.

Notes

This function always returns false for ellipses that are degenerate.

See **ccShape::within()** for more information. See also **isDegenerate()** on page 1307.

perimeter

```
virtual double perimeter() const;
```

Returns the length of the perimeter of this ellipse. See **ccShape::perimeter()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform &X) const;
```

Returns this ellipse mapped by the transformation object *X*.

Parameters

X	The transformation object.

See **ccShape::mapShape()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this ellipse. See **ccShape::reverse()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccEllipseArc2**s. See **ccShape::decompose()** for more information.

ccEllipse2

ccEllipseAnnulus

```
#include <ch_cvl/shapes.h>

class ccEllipseAnnulus : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes an ellipse annulus, a ring shape made up of two ellipses.

Constructors/Destructors

ccEllipseAnnulus

```
ccEllipseAnnulus();

ccEllipseAnnulus(const cc2Vect& c, double r1, double r2);

ccEllipseAnnulus(const ccEllipse2& inner,
  const ccEllipse2& inner);

ccEllipseAnnulus(const ccCircle& inner,
  const ccCircle& outer);
```

- `ccEllipseAnnulus();`

  Default constructor. Constructs a degenerate ellipse annulus.

- `ccEllipseAnnulus(const cc2Vect& c, double r1, double r2);`

  Constructs an ellipse annulus with the specified center and radii.

  Parameters

c	The center of the radius.
r1	The inner radius.
r2	The outer radius.

- `ccEllipseAnnulus(const ccEllipse2& inner, const ccEllipse2& outer);`

  Creates an ellipse annulus from two concentric ellipses.

  Parameters

  inner The inner ellipse.

  outer The outer ellipse.

  Throws

  ccShapesError::BadGeom
  inner and *outer* do not have the same orientation and aspect ration (that is, the ratio of the x to y radii).

  ccShapesError::NotConcentric
  inner and *outer* are not concentric.

  Notes

  This constructor ignores and discards the phase of the *inner* and *outer* ellipses.

- `ccEllipseAnnulus(const ccCircle& inner, const ccCircle& outer);`

  Creates an ellipse annulus from two concentric circles.

  Parameters

  inner The inner circle.

  outer The outer circle.

  Throws

  ccShapesError::NotConcentric
  inner and *outer* are not concentric.

Operators

operator== `bool operator==(const ccEllipseAnnulus& other) const;`

Returns true if this ellipse annulus is equal to another ellipse annulus: it has the same center, inner radius, and outer radius as the other ellipse annulus.

Parameters

other The other ellipse annulus.

operator!=

```
bool operator!=(const ccEllipseAnnulus&) const;
```

Returns true if this ellipse annulus is not equal to another ellipse annulus.

Parameters

other The other ellipse annulus.

Public Member Functions

inner

```
ccEllipse2 inner() const;
```

Returns the inner ellipse.

outer

```
ccEllipse2 outer() const;
```

Returns the outer ellipse.

center

```
const ccPoint& center() const;
```

Returns the center of the ellipse annulus.

innerRadii

```
cc2Vect innerRadii() const;
```

Returns the radii of the inner ellipse.

outerRadii

```
cc2Vect outerRadii() const;
```

Returns the radii of the outer ellipse.

map

```
ccEllipseAnnulus map(const cc2Xform& c) const;
```

Returns an ellipse annulus that is the result of mapping this ellipse annulus with the transformation object *c*.

Parameters

c The transformation object.

Notes

The transform *c* must be nonsingular, and the outer radii of both ellipses must be strictly positive. If both of these conditions are false, use **mapshape()** instead.

ccEllipseAnnulus

degen

```
bool degen() const;
```

Returns true if the ellipse annulus is degenerate, that is if any of the radii of either ellipse is less than or equal to zero.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this ellipse annulus.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For ellipse annuli, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For ellipse annuli, this function always returns true, even for ellipse annuli that are degenerate. See **ccShape::isRegion()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

Returns true if this ellipse annulus is right-handed. See **ccShape::isRightHanded()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For ellipse annuli, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For ellipse annuli, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

This function returns true if **hasTangent()** is true for the inner or outer ellipse. It returns false if **hasTangent()** is false for both ellipses. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For ellipse annuli, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For ellipse annuli, this function always returns false. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this ellipse annulus. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this ellipse annulus to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this ellipse annulus to the given point, as determined by **nearestPoint()**.

Parameters

info	Shape information for this ellipse annulus.
point	The point.

See **ccShape::nearestPerimPos()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.

result Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores ellipse annuli for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this ellipse annulus mapped by *X*.

Parameters

X The transformation object.

Notes

If the transform *X* is singular or either of the outer radii is zero, this function returns a **ccRegionTree**. If both of these conditions are false, this function returns a **ccEllipseAnnulus**.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccEllipseArc2**s. See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if the given point is within this ellipse annulus.

Parameters

p The point.

Notes

This function returns false if either of the underlying ellipses is degenerate.

See **ccShape::within()** for more information.

subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this ellipse annulus over the given perimeter range.

Parameters

info	Shape information for this ellipse annulus.
range	The perimeter range.

See **ccShape::subShape()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

ccEllipseAnnulus

```
ccEllipseAnnulus(const ccEllipse& inner,
  const ccEllipse& outer);
```

Use the constructor that constructs an ellipse annulus from a **ccEllipse2** instead of this version.

innerEllipse

```
const ccEllipse& innerEllipse() const;
```

Use **inner()** instead of this function.

outerEllipse

```
const ccEllipse& outerEllipse() const;
```

Use **outer()** instead of this function.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

ccEllipseAnnulus

ccEllipseAnnulusSection

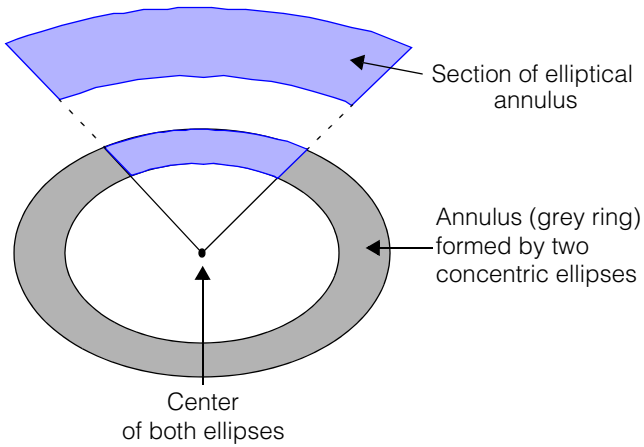
```
#include <ch_cvl/shapes.h>

class ccEllipseAnnulusSection : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a section of an elliptical annulus, as shown in the following diagram.



An elliptical annulus might be used, for example, to draw boundaries around a bar code, serial number, or other mark etched in a curve around the inner ring of a music CD or around the outer ring of a chip wafer. You could then use **cfPolarTransformImage** and **ccPolarSamplingParams** to transform the pixels in the elliptical annulus section image into a rectangular image for easier interpretation by OCR or OCV vision tools.

Constructors/Destructors

ccEllipseAnnulusSection

```
ccEllipseAnnulusSection();

ccEllipseAnnulusSection(const cc2Vect& c, double r1,
  double r2, const ccRadian& phi1, const ccRadian& phi2,
  bool increasingAngle = true, bool centrifugal = true,
  bool normalizePhi2 = true);

ccEllipseAnnulusSection(const ccCircle& c1,
  const ccCircle& c2, const ccRadian& phi1,
  const ccRadian& phi2, bool increasingAngle = true,
  bool centrifugal = true, bool normalizePhi2 = true);

ccEllipseAnnulusSection(const ccEllipse2& c1,
  const ccEllipse2& c2, const ccRadian& phi1,
  const ccRadian& phi2, bool increasingAngle = true,
  bool centrifugal = true, bool normalizePhi2 = true);

ccEllipseAnnulusSection(const ccEllipseAnnulus& ann,
  const ccRadian& phi1, const ccRadian& phi2,
  bool increasingAngle = true, bool centrifugal = true,
  bool normalizePhi2 = true);

ccEllipseAnnulusSection(const ccAnnulus& ann,
  const ccRadian& phi1, const ccRadian& phi2,
  bool increasingAngle = true, bool centrifugal = true,
  bool normalizePhi2 = true);
```

- `ccEllipseAnnulusSection();`

  The default constructor creates a degenerate elliptical annulus section.

- `ccEllipseAnnulusSection(const cc2Vect& c, double r1, double r2, const ccRadian& phi1, const ccRadian& phi2, bool increasingAngle = true, bool centrifugal = true, bool normalizePhi2 = true);`

  Creates an elliptical annulus section given a center, radii, start and end angles.

  Parameters

c	The center of the ellipse annulus section.
r1	The inner radius.
r2	The outer radius.

phi1 The starting angle of the ellipse annulus section.

phi2 The ending angle of the ellipse annulus section.

increasingAngle True if the angle increases. Used for polar unwrapping.

centrifugal True if unwrapping from the center to the outer edge. Used for polar unwrapping.

normalizePhi2 True to ensure that *phi2* is greater than *phi1*.

Throws

ccShapesError::DegenerateShape
phi2 < *phi1* and *normalizePhi2* is false.

Notes

See *Notes* on page 1324 for a discussion of measuring the *phi1* and *phi2* angles.
See *Notes* on page 1329 for illustrations of the effects of the *increasingAngle* and *centrifugal* arguments.

- ```
  ccEllipseAnnulusSection(const ccCircle& c1,
    const ccCircle& c2, const ccRadian& phi1,
    const ccRadian& phi2, bool increasingAngle = true,
    bool centrifugal = true, bool normalizePhi2 = true);
  ```

  Creates an ellipse annulus section from two concentric circles.

  Parameters

  c1 The inner circle.

  c2 The outer circle.

  phi1 The starting angle of the ellipse annulus section.

  phi2 The ending angle of the ellipse annulus section.

  increasingAngle True if the angle increases. Used for polar unwrapping.

  centrifugal True if unwrapping from the center to the outer edge. Used for polar unwrapping.

  normalizePhi2 True to ensure that *phi2* is greater than *phi1*.

  Throws

  ccShapesError::DegenerateShape
  phi2 < *phi1* and *normalizePhi2* is false.

  ccShapesError::NotConcentric
  c1 and *c2* are not concentric
  ```

Notes

See *Notes* on page 1324 for a discussion of measuring the *phi1* and *phi2* angles. See *Notes* on page 1329 for illustrations of the effects of the *increasingAngle* and *centrifugal* arguments.

- ```
  ccEllipseAnnulusSection(const ccEllipse2& c1,
    const ccEllipse2& c2, const ccRadian& phi1,
    const ccRadian& phi2, bool increasingAngle = true,
    bool centrifugal = true, bool normalizePhi2 = true);
  ```

  Creates an ellipse annulus section from two concentric ellipses.

  Parameters

c1	The inner ellipse.
c2	The outer ellipse.
phi1	The starting angle of the ellipse annulus section.
phi2	The ending angle of the ellipse annulus section.
increasingAngle	True if the angle increases. Used for polar unwrapping.
centrifugal	True if unwrapping from the center to the outer edge. Used for polar unwrapping.
normalizePhi2	True to ensure that *phi2* is greater than *phi1*.

  Throws

  ccShapesError::NotConcentric
  The two ellipses, *c1* and *c2*, do not have the same center, ratio of *x* and *y* radii, and orientation.

  ccShapesError::DegenerateShape
  phi2 is less than *phi1* and *normalizePhi2* is false.

  Notes

  This constructor ignores and discards the phase of the inner and outer ellipses.

  See *Notes* on page 1324 for a discussion of measuring the *phi1* and *phi2* angles. See *Notes* on page 1329 for illustrations of the effects of the *increasingAngle* and *centrifugal* arguments.
  ```

- ```
  ccEllipseAnnulusSection(const ccEllipseAnnulus& ann,
    const ccRadian& phi1, const ccRadian& phi2,
    bool increasingAngle = true, bool centrifugal = true,
    bool normalizePhi2 = true);
  ```

  Creates an ellipse annulus section from an ellipse annulus.

  Parameters

ann	The ellipse annulus.
phi1	The starting angle of the ellipse annulus section.
phi2	The ending angle of the ellipse annulus section.
increasingAngle	True if the angle increases. Used for polar unwrapping.
centrifugal	True if unwrapping from the center to the outer edge. Used for polar unwrapping.
normalizePhi2	True to ensure that *phi2* is greater than *phi1*.

  Throws

  ccShapesError::DegenerateShape
  phi2 < *phi1* and *normalizePhi2* is false.

  Notes

  See *Notes* on page 1324 for a discussion of measuring the *phi1* and *phi2* angles.
  See *Notes* on page 1329 for illustrations of the effects of the *increasingAngle* and *centrifugal* arguments.

- ```
  ccEllipseAnnulusSection(const ccAnnulus& ann,
    const ccRadian& phi1, const ccRadian& phi2,
    bool increasingAngle = true, bool centrifugal = true,
    bool normalizePhi2 = true);
  ```

  Creates an ellipse annulus section from an annulus.

  Parameters

ann	The annulus.
phi1	The starting angle of the ellipse annulus section.
phi2	The ending angle of the ellipse annulus section.
increasingAngle	True if the angle increases. Used for polar unwrapping.
centrifugal	True if unwrapping from the center to the outer edge. Used for polar unwrapping.
normalizePhi2	True to ensure that *phi2* is greater than *phi1*.

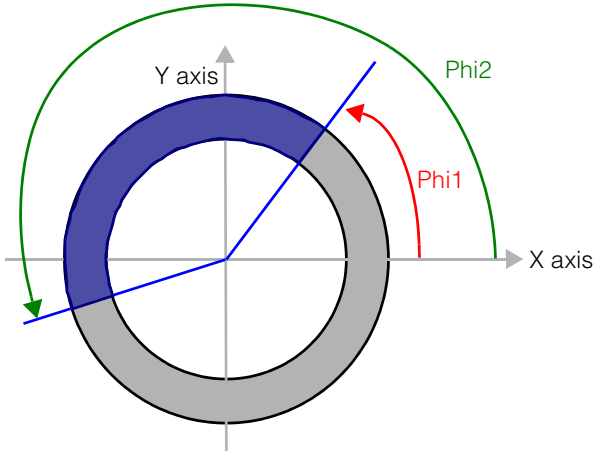
Throws

ccShapesError::DegenerateShape
phi2 < *phi1* and *normalizePhi2* is false.

Notes

The starting angle *phi1* of the section, and the ending angle *phi2,* are measured with respect to the section projected onto a unit circle. Angles are measured from the X axis of the unit circle.

In the case of an annular section of two concentric circles, the starting angle is simply the angle from the X axis to the circle radius that marks the beginning of the section. Likewise, the ending angle *phi2* is the angle from the X axis to the radius that marks the end of the section. This case is shown in the following figure.



In the general case of an annular section of two concentric ellipses, the **ccEllipse** object that represents the outer ellipse includes a **ccEllipseUnit** structure. This structure defines the transformation from ellipse coordinates to unit circle coordinates. You can use this transformation to calculate the starting and ending angle values. For a given point *p* on the section's outer ellipse, its angle *phi* in the unit coordinate system is computed by the following expression:

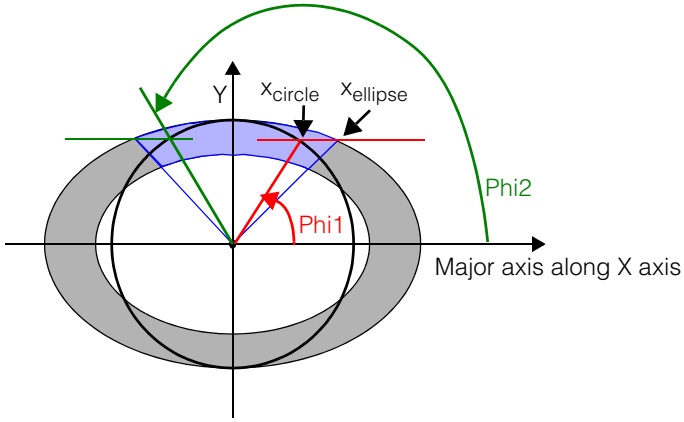
```
ccEllipse2  outerEllipse;
cc2Vect     center; // default constructor creates as (0,0)
ccPoint     p;

(outerEllipse.unit().U_ * (p - center)).angle();
```

Apply the above expression in turn to each endpoint *p* of the outer elliptical arc of the section to yield the values of *phi1* and *phi2* for the section.

If you do not have a **ccEllipse2** object to start from, you can obtain approximate *phi1* and *phi2* angles from the projection method illustrated in the following diagram. Draw a unit reference circle whose radius is one-half of the minor axis of the outer ellipse; this reference circle fits just inside the outer ellipse. Draw a line parallel to the major axis, and passing through the endpoint, $x_{ellipse}$, of the outer elliptical arc of the section. The point where this line intersects the inner reference circle, x_{circle}, is the projection of $x_{ellipse}$ onto the circle. The section's starting angle *phi1* is measured from the major axis to the line between the center and x_{circle}.

Likewise the ending angle *phi2* is measured from the major axis to the line formed between the center and the projection of the other outer arc endpoint onto the reference circle.



This projection method and the diagram above assume that the transformation of the outer ellipse onto the unit reference circle is not reflected (that is, it presumes that **outerEllipse.geom().reverseAngleDir_** is false, the default condition).

Operators

operator==

```
bool operator==(const ccEllipseAnnulusSection& e) const;
```

Returns true if this ellipse annulus section is equal to another ellipse annulus section: it has the same center, inner radius, outer radius, start angle, and end angle as the other ellipse annulus.

Parameters

e	The other ellipse annulus section.

operator!=

```
bool operator!=(const ccEllipseAnnulusSection& e) const;
```

Returns true if this ellipse annulus section is not equal to another ellipse annulus section.

Parameters

e	The other ellipse annulus section.

Public Member Functions

ellipseAnnulus

```
const ccEllipseAnnulus& ellipseAnnulus() const;
```

Returns the complete ellipse annulus that this object is a section of.

center

```
const ccPoint& center() const;
```

Returns the center of the ellipse annulus section.

phi1

```
const ccRadian &phi1() const;
void phi1(const ccRadian &phi1);
```

- `const ccRadian &phi1() const;`

  Returns the starting angle of the ellipse annulus section.

- `void phi1(const ccRadian &phi1);`

  Sets the starting angle of the ellipse annulus section.

  Parameters

phi1	The starting angle.

  Notes

  See *Notes* on page 1324 for a discussion of how starting angles are measured.

phi2

```
const ccRadian &phi2() const;
void phi2(const ccRadian &phi2);
```

- `const ccRadian &phi2() const;`

  Returns the ending angle of the ellipse annulus section.

- `void phi2(const ccRadian &phi2);`

  Sets the ending angle of the ellipse annulus section.

  Parameters
  phi2 The ending angle.

  Notes
  See *Notes* on page 1324 for a discussion of how ending angles are measured.

increasingAngle

```
bool increasingAngle() const;
void increasingAngle(bool increasingAngle);
```

- `bool increasingAngle() const;`

  Return true if the angle is increasing. This information is used by the polar unwrapping functions.

- `void increasingAngle(bool increasingAngle);`

  Sets whether the angle is increasing.

  Parameters
  increasingAngle True if the angle increases. Used for polar unwrapping.

  Notes
  See *Notes* on page 1329 for illustrations of the effect of *increasingAngle*.

centrifugal

```
bool centrifugal() const;
void centrifugal(bool centrifugal);
```

- `bool centrifugal() const;`

  Returns true if unwrapping from the center to the outer edge. Used for polar unwrapping.

- `void centrifugal(bool centrifugal);`

Sets whether unwrapping from the center to the outer edge. Used for polar unwrapping.

Parameters

centrifugal True if unwrapping from the center to the outer edge. Used for polar unwrapping.

Notes

See *Notes* on page 1329 for illustrations of the effect of *centrifugal*.

inner `ccEllipse2 inner() const;`

Returns the inner ellipse.

outer `ccEllipse2 outer() const;`

Returns the outer ellipse.

map `ccEllipseAnnulusSection map(const cc2Xform& c) const;`

Returns an ellipse annulus section that is the result of mapping this ellipse annulus section with the transformation object *c*.

Parameters

c The transformation object.

Throws

ccShapesError::BadCoeff
The mapping is not a pure rotation, translation, and scale mapping.

Notes

Avoid using this method with transforms that have a negative determinant. With these kind of transforms, this method does *not* correctly maintain the correspondence between points of the ellipse annulus section. For example, if *X* is a transofrm with a negative determinant and *eas* is an ellipse annulus section, it will *not* necessarily be the case that **eas.map(X).cornerPo()** approximately equals **X.mapPoint(eas.cornerPo())**, as might be expected. Use **mapShape()** instead of this function where possible, as it does not exhibit this problem.

degen `bool degen() const;`

Returns true if the ellipse annulus section is degenerate, that is if any of the radii of either ellipse is less than or equal to zero.

cornerPo `ccPoint cornerPo() const;`

Returns the corner point P_o of the ellipse annulus section. Refer to the illustrations below.

cornerPx `ccPoint cornerPx() const;`

Returns the corner point P_x of the ellipse annulus section.

cornerPy `ccPoint cornerPy() const;`

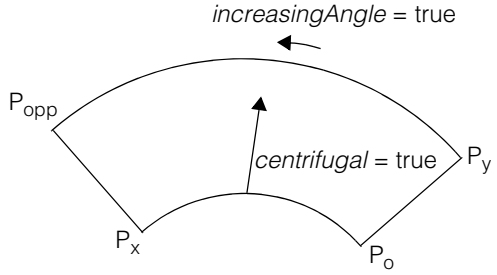
Returns the corner point P_y of the ellipse annulus section.

cornerPopp `ccPoint cornerPopp() const;`

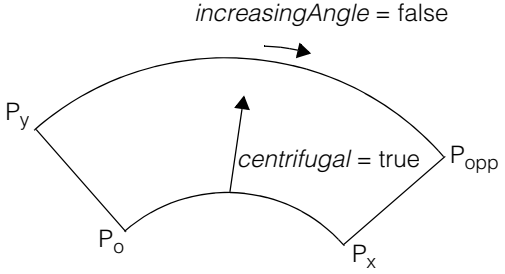
Gets the corner point P_{opp} of the ellipse annulus section.

Notes

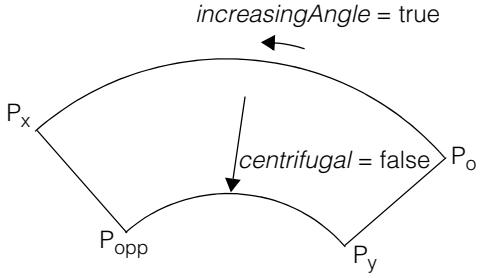
The **corner*** functions refer to the points shown in the following figures. The first figure shows the default case, where *increasingAngle* and *centrifugal* are true:



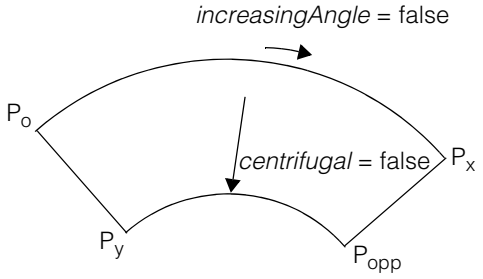
The points are reversed left-to-right if *increasingAngle* is false:



The points are reversed up-to-down if *centrifugal* is false:



Finally, the points are reversed both ways if both *increasingAngle* and *centrifugal* are false:



mapFromUnitSq `ccPoint mapFromUnitSq(const ccPoint &pt) const;`

Maps the given point from the unit square to the ellipse annulus section.

Parameters

pt The point.

mapToUnitSq `ccPoint mapToUnitSq(const ccPoint &pt) const;`

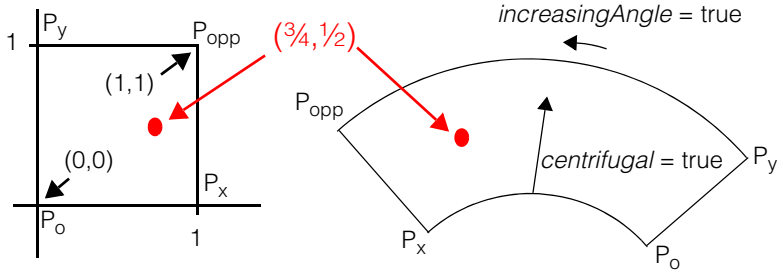
Maps the given point to the unit square from the ellipse annulus section.

Parameters

pt The point.

Notes

The **map*UnitSq** functions map a point to and from anywhere on or within an ellipse annulus section to a reference unit square with height and width equal to 1. A section and unit square are shown in the following figure:



In the unit square projection, the X axis corresponds to the angle from *phi1* at which a point in the annular section is found. The Y axis corresponds to the radius of the point. Thus, a point, illustrated in red in the figure, about 3/4 of the way over in the direction of the *increasingAngle* vector, and about 1/2 way into the section in the direction of the *centrifugal* vector, would be mapped as shown in the reference unit square.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this ellipse annulus section.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For ellipse annulus sections, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For ellipse annulus sections, this function always returns true, even for ellipse annulus sections that are degenerate. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For ellipse annulus sections, this function always returns true. See **ccShape::isFinite()** for more information.

ccEllipseAnnulusSection

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For ellipse annulus sections, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

This function returns true if the perimeter of this ellipse annulus section is positive. It returns false if the perimeter is zero. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For ellipse annulus sections, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible `virtual bool isReversible() const;`

For ellipse annulus sections, this function always returns false. See **ccShape::reverse()** for more information.

isRightHanded `virtual bool isRightHanded() const;`

Returns true if this ellipse annulus section is right-handed. See **ccShape::isRightHanded()** for more information.

boundingBox `virtual ccRect boundingBox() const;`

Returns the smallest rectangle that encloses this ellipse annulus section. See **ccShape::boundingBox()** for more information.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const;`

Returns the nearest point on the boundary of this ellipse annulus section to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p	The point.

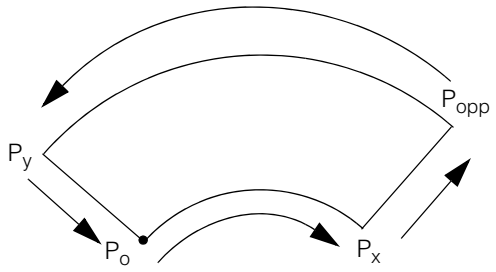
See **ccShape::nearestPoint()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape. The corner points of the ellipse annulus section are traversed in the following order:

1. P_o
2. P_x
3. P_{opp}
4. P_y
5. P_o



Sampling Starts and Ends at P_o

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores ellipse annulus sections for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this ellipse annulus section mapped by *X*.

Parameters

X	The transformation object.

Notes

If the transform *X* is singular, this function returns a closed **ccContourTree**. Otherwise, it returns a **ccEllipseAnnulusSection**.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Decomposes this **ccEllipseAnnulusSection** into a closed **ccContourTree** with four children in the following order:

1. An arc from P_o to P_x
2. A line segment from P_x to P_{opp}
3. An arc from P_{opp} to P_y
4. A line segment from P_y to P_o

See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if the given point is within this ellipse annulus section.

Parameters

p The point.

Notes

This function returns false if either of the underlying ellipses is degenerate.

See **ccShape::within()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

ccEllipseAnnulusSection

```
ccEllipseAnnulusSection(const ccEllipse& c1,
  const ccEllipse2& c2, const ccRadian& phi1,
  const ccRadian& phi2, bool increasingAngle = true,
  bool centrifugal = true, bool normalizePhi2 = true);
```

Use the constructor that uses **ccEllipse2** references as its first two arguments instead of this version.

innerEllipse

```
const ccEllipse &innerEllipse() const;
```

Use **inner()** instead of this function.

outerEllipse `const ccEllipse &outerEllipse() const;`

Use **outer()** instead of this function.

distToPoint `double distToPoint(const cc2Vect& v) const;`

Use **distanceToPoint()** instead of this function.

encloseRect `ccRect encloseRect() const;`

Use **boundingBox()** instead of this function.

ccEllipseAnnulusSection

ccEllipseArc

```
#include <ch_cvl/oldshape.h>

class ccEllipseArc : public ccEllipse;
```

This class is deprecated. Use the **ccEllipseArc2** class instead.

ccEllipseArc2

```
#include <ch_cvl/ellipse.h>

class ccEllipseArc2 : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccEllipseArc2** class represents arbitrary 2D elliptical arcs. An elliptical arc is represented as an underlying ellipse plus a parameter range that specifies that portion of the ellipse included in the arc. The parameter range is specified by a starting angle and an angular span. Together, these values specify the range of *phi* values of the underlying ellipse that lie on the elliptical arc.

Note The *phase* and *handedness* of the underlying ellipse of an ellipse arc affect the range of *phi* values of the underlying ellipse that are on the elliptical arc. See **ccEllipse2** for more information.

The angular span may be positive or negative. Changing the sign of the angle range changes the direction along the ellipse in which the ellipse arc lies relative to the starting point. Flipping the *isRightHanded* flag of the underlying ellipse has the same effect. The angular span may have a magnitude greater than 2π radians, so it is possible to specify multiple revolutions around the ellipse. The **ccEllipse2** class provides methods for manipulating the underlying ellipse as well as the angle range of the arc.

Tangents

The tangent direction along an ellipse arc is always in the direction from the start point, at parameter value **phiStart()**, toward the end point, at parameter value **phiEnd()**. This means that if **phiSpan()** is positive, the tangent directions are aligned with the tangent directions of the underlying ellipse, and if **phiSpan()** is negative, the tangent directions are opposite those of the underlying ellipse.

Constructors/Destructors

ccEllipseArc2

```
ccEllipseArc2(const ccEllipse2 &ell = ccEllipse2());

ccEllipseArc2(const ccEllipse2 &ell,
  const ccRadian &phiStart, const ccRadian &phiSpan);

explicit ccEllipseArc2(const ccEllipseArc &ellArc);
```

- `ccEllipseArc2(const ccEllipse2 &ell = ccEllipse2());`

  Conversion constructor. Constructs an ellipse arc with the given underlying ellipse and an angle range of 0 through $\pi/2$ radians.

  Parameters

  ell — The underlying ellipse. The default **ccEllipse2** is a right-handed unit circle, centered at the origin, with zero angle and phase.

- `ccEllipseArc2(const ccEllipse2 &ell, const ccRadian &phiStart, const ccRadian &phiSpan);`

  Constructs an ellipse arc with the given underlying ellipse, starting angle, and angular span.

  Parameters

  ell — The underlying ellipse.

  phiStart — The starting angle. This parameter is only significant modulo 2π radians.

  phiSpan — The angular span. his parameter is significant over all values, positive and negative. Values of *phiSpan* with a magnitude exceeding 2π radians specify arcs comprising more than one revolution around the underlying ellipse.

- `explicit ccEllipseArc2(const ccEllipseArc &ellArc);`

  Conversion constructor. Constructs a **ccEllipseArc2** from a deprecated **ccEllipseArc**. The underlying ellipse of the constructed **ccEllipseArc2** will have zero phase. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

  ellArc — The old-style ellipse arc.

Notes

This constructor is provided to support legacy code that might need to convert an old-style **ccEllipseArc** to a new-style **ccEllipseArc2**. It is not recommended as a way of constructing a **ccEllipseArc2** in new code.

Operators

operator==

```
bool operator==(const ccEllipseArc2 &rhs) const;
```

Returns true if an only if this ellipse arc is equal to *rhs*. This method tests for exact equality of all internal data members.

Parameters

rhs The other ellipse arc.

operator!=

```
bool operator!=(const ccEllipseArc2 &rhs) const;
```

Returns true if this ellipse arc is not equal to *rhs*.

Parameters

rhs The other ellipse arc.

Public Member Functions

ellipse

```
const ccEllipse2 &ellipse() const;
void ellipse(const ccEllipse2 &ell);
```

- ```
  const ccEllipse2 &ellipse() const;
  ```

  Gets the underlying ellipse of this ellipse arc.

- ```
  void ellipse(const ccEllipse2 &ell);
  ```

  Sets the underlying ellipse of this ellipse arc.

  Parameters

  ell The underlying ellipse.

phiStart

```
ccRadian phiStart() const;
void phiStart(const ccRadian &s);
```

- `ccRadian phiStart() const;`

  Gets the starting angle of this ellipse arc.

- `void phiStart(const ccRadian &s);`

  Sets the starting angle of this ellipse arc.

  Parameters

  s The starting angle.

  Notes

  The starting angle is only significant modulo 2π radians, although neither the getter nor the setter force its value to lie in this range.

phiSpan

```
ccRadian phiSpan() const;
void phiSpan(const ccRadian &s);
```

- `ccRadian phiSpan() const;`

  Gets the angular span of this ellipse arc.

- `void phiSpan(const ccRadian &s);`

  Sets the angular span of this ellipse arc.

  Parameters

  s The angular span.

  Notes

  The angle span is significant over all values, positive and negative. Values of *phiSpan* with magnitude exceeding 2π radians specify arcs comprising more than one revolution around the underlying ellipse.

phiEnd

```
ccRadian phiEnd() const;
```

Gets the ending angle of this ellipse arc, equal to *phiStart()* + *phiSpan()*.

map

```
ccEllipseArc2 map(const cc2XformLinear &X) const;
```

Parameters

Returns this ellipse arc mapped by the transformation object *X*.

Parameters

X	The transformation object.

Notes

This method operates by calling the corresponding method on the underlying ellipse and leaving the arc angle range unchanged.

See **ccEllipse2::map()** for details.

translate

```
ccEllipseArc2 translate(const cc2Vect &offset) const;
```

Returns this ellipse arc translated by the vector *offset*.

Parameters

offset	The vector containing the offset value.

Notes

This method operates by calling the corresponding method on the underlying ellipse and leaving the arc angle range unchanged.

See **ccEllipse2::translate()** for details.

rotate

```
ccEllipseArc2 rotate(const ccRadian &theta) const;
```

Returns this ellipse arc rotated about its center by the angle *theta*.

Parameters

theta	The angle.

Notes

This method operates by calling the corresponding method on the underlying ellipse and leaving the arc angle range unchanged.

See **ccEllipse2::rotate()** for details.

scale

```
ccEllipseArc2 scale(const cc2Vect &mag) const;
```

Returns this ellipse arc scaled by the vector *mag* about its center and along the directions of its axes of symmetry.

Parameters

mag The vector containing the magnification value.

Throws

ccShapesError::BadParams
Either component of *mag* is negative.

Notes

This method operates by calling the corresponding method on the underlying ellipse and leaving the arc angle range unchanged.

See **ccEllipse2::scale()** for details.

isInSpan

```
bool isInSpan(const ccRadian &phi) const;
```

Returns true if and only if *phi* corresponds to a point on this ellipse arc.

Parameters

phi An angle that specifies a point that may or may not be on this ellipse arc.

Notes

The values **phiStart()** and **phiEnd()** together specify a half-open range if **phiSpan()** is positive. In other words, for a positive angle span, **isInSpan()** is true for the starting angle, returned by **phiStart()**, and false for the ending angle, returned by **phiEnd()**.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this ellipse arc.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For ellipse arcs, this function always returns true. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For ellipse arcs, this function always returns false. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For ellipse arcs, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For ellipse arcs, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

For ellipse arcs, this function returns true if **hasTangent()** is true for the underlying ellipse and **phiSpan()** is not zero. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For ellipse arcs, this function always returns true. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For ellipse arcs, this function always returns true. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this ellipse arc. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on this ellipse arc to the given point. See **ccShape::nearestPoint()** for more information.

Parameters

p	The point.

perimeter

```
virtual double perimeter() const;
```

Returns the perimeter of this ellipse arc. See **ccShape::perimeter()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this ellipse arc to the given point, as determined by **nearestPoint()**.

Parameters

info Shape information for this ellipse arc.

point The point.

Notes

The returned position always corresponds to an angular position within 2π radians from the start angle, even when the magnitude of **phiSpan()** exceeds 2π.

See **ccShape::nearestPerimPos()** for more information.

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Returns the net signed angle through which the tangent vector rotates along this ellipse arc from the start point to the end point.

Throws

ccShapesError::NoTangent
hasTangent() is false for this ellipse arc.

See **ccShape::tangentRotation()** for more information.

windingAngle

```
virtual ccRadian windingAngle(const cc2Vect &p) const;
```

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve.

Parameters

p A point.

See **ccShape::windingAngle()** for more information.

startPoint

```
virtual cc2Vect startPoint() const;
```

Returns the starting point of this ellipse arc. See **ccShape::startPoint()** for more information.

endPoint

```
virtual cc2Vect endPoint() const;
```

Returns the ending point of this ellipse arc. See **ccShape::endPoint()** for more information.

startAngle

```
virtual ccRadian startAngle() const;
```

Returns the starting angle of this ellipse arc.

Throws

ccShapesError::NoTangent
hasTangent() is false for this ellipse arc.

See **ccShape::startAngle()** for more information.

endAngle

```
virtual ccRadian endAngle() const;
```

Returns the ending angle of this ellipse arc.

Throws

ccShapesError::NoTangent
hasTangent() is false for this ellipse arc.

See **ccShape::endAngle()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this ellipse arc. See **ccShape::reverse()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this ellipse arc.

Parameters

params	Parameters object specifying details of how the sampling should be done.
result	Result object to which position and tangent chains are stored.

Notes

If **params.computeTangents()** is true, this function ignores ellipse arcs for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform &X) const;
```

Returns this ellipse arc mapped by the transformation object *X*.

Parameters

X	The transformation object.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a clone of this ellipse arc. As an ellipse arc is already a decomposed shape, this method is equivalent to **clone()** for ellipse arcs. See **ccShape::decompose()** for more information.

subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape on the perimeter of this ellipse arc within the given range.

Parameters

info	Shape information for this ellipse arc.
range	The perimeter range.

See **ccShape::subShape()** for more information.

ccEllipseFitDefs

```
#include <ch_cvl/fit.h>

class ccEllipseFitDefs;
```

A name space that holds enumerations and constants used with the Ellipse Fitting tool.

Enumerations

fit_mode

```
enum fit_mode
```

This enumeration defines types of fitting supported by the Ellipse Fitting tool.

Value	Meaning
eLeastSquares = 0	Fits an ellipse by minimizing the least squares error.
eLeastSquaresUseExpectedOrientation = 1	Fits an ellipse whose orientation you specify by minimizing the least squares error.
eLeastSquaresUseAccurateRMSError = 2	Fits an ellipse by minimizing the least squares error and subtracting the accurate RMS measurement error.
eLeastSquaresUseExpectedOrientationAndAccurateRMSError = 3	Fits an ellipse whose orientation you specify by minimizing the least squares error and subtracting the accurate RMS measurement error.
kDefaultFitMode = eLeastSquares	

ccEllipseFitDefs

ccEllipseFitParams

```
#include <ch_cvl/fit.h>

class ccEllipseFitParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds the Ellipse Fitting tool parameters. See **cfEllipseFit()**.

Constructors/Destructors

ccEllipseFitParams

```
ccEllipseFitParams(
  enum ccEllipseFitDefs::fit_mode fitMode =
                          ccEllipseFitDefs::kDefaultFitMode,
  const ccRadian &orientation = ccRadian(0),
  c_UInt32 numIgnore=0,
  double threshold=HUGE_VAL);
```

Constructs an ellipse fit parameters object

Parameters

fitMode — The ellipse fit mode. Must be one of the following:

eLeastSquares; the tool finds the ellipse axes.

eLeastSquaresUseExpectedOrientation; you provide the ellipse orientation (see *orientation* below). Can cause the tool to run faster but may produce a less accurate found position

orientation — The expected orientation of the ellipse. You can specify either the ellipse major axis orientation or the minor axis orientation. The fitter will work with either axis.

This parameter is only used when *fitMode = eLeastSquaresUseExpectedOrientation*.

numIgnore — The number of outlying points to ignore. The tool always discards this number of points from the point set used to find the ellipse.

threshold The maximum RMS error for a valid fit. If the fitted ellipse has an RMS error greater than *threshold*, it is not considered a valid fit.

Throws

ccEllipseFitDefs::BadParams
If *threshold* < 0

Operators

operator==

```
bool operator== (const ccEllipseFitParams& that) const;
```

True if **this* equals *that*; false otherwise.

Two **ccEllipseFitParams** objects are considered equal if and only if their fitMode, radius, numIgnore, and threshold values are each equal.

Parameters

that The **ccEllipseFitParams** object to compare with this object.

Public Member Functions

fitMode

```
enum ccEllipseFitDefs::fit_mode fitMode();

void fitMode(enum ccEllipseFitDefs::fit_mode fitMode);
```

See the CTOR description.

- `enum ccEllipseFitDefs::fit_mode fitMode();`

  Returns the ellipse fitting mode.

- `void fitMode(enum ccEllipseFitDefs::fit_mode fitMode);`

  Sets a new ellipse fit mode.

  Parameters

  fitMode The new ellipse fit mode. Must be one of the *ccEllipseFitDefs::fit_mode* enums.

orientation

```
ccRadian orientation() const;

void orientation(const ccRadian &orientation);
```

See *orientation* in the CTOR description.

- `ccRadian orientation() const;`

  Returns the current ellipse orientation angle specification.

- `void orientation(const ccRadian &orientation);`

  Sets a new ellipse orientation angle specification. This expected orientation may correspond to the major axis or the minor axis of the ellipse.

  Parameters
  orientation The new orientation angle.

numIgnore

```
c_UInt32 numIgnore() const;
void numIgnore(c_UInt32 numIgnore);
```

The number of outlying points to ignore. The tool always discards this number of points from the point set used to find the ellipse.

- `c_UInt32 numIgnore() const;`

  Returns the current number specified.

- `void numIgnore(c_UInt32 numIgnore);`

  Sets a new number of outlying points to ignore.

  Parameters
  numIgnore The number of points to ignore.

threshold

```
double threshold() const;
void threshold(double threshold);
```

The maximum RMS error for a valid fit. If the fitted ellipse has an RMS error greater than *threshold*, it is not considered a valid fit.

- `double threshold() const;`

  Returns the current threshold value.

- `void threshold(double threshold);`

  Sets a new threshold value.

Parameters

threshold The new threshold value.

Throws

ccEllipseFitDefs::BadParams
If *threshold* < 0.

ccEllipseFitResults

```
#include <ch_cvl/fit.h>

class ccEllipseFitResults;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds the results from the Ellipse Fitting tool. See **cfEllipseFit()**.

Constructors/Destructors

ccEllipseFitResults

```
ccEllipseFitResults()
```

Default constructor. **found()** = false, **error()** = 0, **time()** = 0.

Operators

operator==

```
bool operator== (const ccEllipseFitResults& that) const;
```

True if **this* equals *that*; false otherwise

Two **ccEllipseFitResults** objects are considered equal if and only if their found, ellipse, error, runParams, outliers, and time values are equal.

Parameters

that The **ccEllipseFitResults** object to compare with this object.

Public Member Functions

found

```
bool found() const;
```

Returns true if an ellipse was found when this object was used by the Ellipse Fitting tool. Returns false otherwise.

reset `void reset();`

Reset this result object to its default-constructed state (unfound).

ellipse2 `const ccEllipse2 &ellipse2() const;`

Returns the fitted ellipse.

error `double error() const;`

Returns the measured RMS error between the input points and the fitted ellipse when the Ellipse Fitting tool was run using this result object. The default value is 0

runParams `const ccEllipseFitParams &runParams() const;`

Returns the run parameters used when the Ellipse Fitting tool was run using this results object.

outliers `const cmStd vector<c_UInt32> &outliers() const;`

Returns the Indices of the ignored data points when the Ellipse Fitting tool was run using this results object. The number of ignored points equals **ccEllipseFitParams::numIgnore()** in the run parameters used.

time `double time() const;`

Returns the time (in milliseconds) used to compute this result. This is the **cfEllipseFit()** run time.

Deprecated Members

The following functions are deprecated and are provided for backward compatibility only.

ellipse `const ccEllipse &ellipse() const;`

Returns the fitted ellipse.

ccEncoderControlProp

```
#include <ch_cvl/prop.h>

class ccEncoderControlProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class provides functions to control a specified encoder connected to a Cognex frame grabber.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccEncoderControlProp

```
ccEncoderControlProp();
```

Creates a new encoder control property not associated with any FIFO.

Public Member Functions

zeroCounter

```
void zeroCounter();
```

If this property is associated with a FIFO that uses an encoder, the encoder's absolute position counter is reset to zero. A subsequent call to **currentEncoderCount()** will return zero.

Notes

If the encoder is currently in use for an acquisition, this function waits for the acquisition to complete before resetting the counter.

Notes

On the MVS-8600 platform, there is no absolute encoder position counter available. There is however, a counter that stores the count of any backward encoder pulses detected by the hardware when the encoder is moving in the opposite direction to the one expected for acquisition. If there are stored backward count pulses, new acquisition will not begin until the backward motion is compensated by forward motion. Calling **zeroCounter()** will clear this backward encoder pulse counter and make acquisition begin immediately without waiting for encoder compensation

zeroCounterWhenTriggered

```
void zeroCounterWhenTriggered(bool zeroWhenTriggered);

bool zeroCounterWhenTriggered() const;
```

- `void zeroCounterWhenTriggered(bool zeroWhenTriggered);`

  Sets whether the absolute position counter is reset to zero when the FIFO associated with this property receives an external trigger. The default setting is false.

  This functionality is not supported by all line scan platforms.

  Parameters

  zeroWhenTriggered

  True if the counter should be reset on a trigger; false otherwise.

- `bool zeroCounterWhenTriggered() const;`

  Returns true if the absolute position counter is reset to zero when the FIFO associated with this property receives an external trigger; false otherwise.

currentEncoderCount

```
c_Int32 currentEncoderCount();
```

Returns the current encoder count if this property is associated with a FIFO that has an encoder.

Notes

On the MVS-8600 platform, there is no absolute encoder position counter available. Therefore, this function returns the product of the currently acquiring line number and the steps per line encoder setting.

The value returned may not be accurate if the encoder travels backward during the acquisition. Also, when a test encoder is being used the steps per line setting is ignored, so the currently acquiring line number will be returned.

Calling **currentEncoderCount()** with no acquisition in progress returns a 0.

ccEncoderControlProp

ccEncoderProp

```
#include <ch_cvl/prop.h>

class ccEncoderProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class controls frame grabber-specific devices for interfacing to an external encoder on hardware that supports an encoder interface such as the CVM11. In most cases, you will want to use this class to set the steps-per-line property for a line scan camera (see **stepsPerLine()** on page 1208).

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccEncoderProp

```
ccEncoderProp();
```

Creates a new encoder property not associated with any FIFO.

Enumerations

ceEncoderResolution

```
enum ceEncoderResolution;
```

This enumeration specifies the choices for the number of counts to expect from the encoder you are using in your application.

Notes

See **encoderResolution** on page 1371 for a detailed description of how encoder pulses are derived.

Value	Meaning
e1xEncoderResolution = 0	The encoder pulse from a full clock cycle produces one count. The default.
e2xEncoderResolution = 1	The encoder pulse from a full clock cycle produces two counts.
e4xEncoderResolution = 2	The encoder pulse from a full clock cycle produces four counts.

Public Member Functions

useTestEncoder

```
void useTestEncoder(bool use);
bool useTestEncoder() const;
```

- `void useTestEncoder(bool use);`

Specifies whether to use signals from the test encoder on the CVM or the actual encoder that this property is associated with. The default value is false.

Parameters

use If true, use the test encoder; if false, use the actual encoder.

Notes

This property affects only the encoder that corresponds to the camera port of the FIFO that this property object is associated with.

On the MVS-8120/CVM11 platform, when using the test encoder, be sure to use **stepsPerLine()** on page 1208 appropriately. If you are acquiring 2048-pixel wide images, the minimum value to use is 34 (for 1024-pixel wide images, the value is 18). Any value smaller than that will result in **isTooFastEncoder()** failures.

On the MVS-8600 and MVS-8600e platforms, there is no test encoder available. When this property is enabled, the hardware free runs and acquires lines as fast as possible without using an encoder input. The **stepsPerLine()** setting is ignored. To introduce a delay between acquisitions of successive lines, you can specify a positive trigger delay value. See the **ccTriggerFilterProp** reference page.

- `bool useTestEncoder() const;`

  Returns true if using the test encoder on the CVM; false if using the actual encoder.

positiveTestEncoderDirection

```
void positiveTestEncoderDirection(bool isPositive);

bool positiveTestEncoderDirection() const;
```

- `void positiveTestEncoderDirection(bool isPositive);`

  Sets whether the test encoder increments or decrements its position counter. The default value is true: the test encoder increments.

  Parameters

  isPositive — If true, the test encoder increments the position counter; if false, the test encoder decrements the position counter.

- `bool positiveTestEncoderDirection() const;`

  Returns true if the test encoder increments the position counter; false if it decrements the position counter.

encoderTriggerEnabled

```
void encoderTriggerEnabled(bool useXEncoder,
  bool useYEncoder=false);

bool encoderTriggerEnabled() const;
```

- `void encoderTriggerEnabled(bool useXEncoder, bool useYEncoder=false);`

  Specifies whether to use an encoder instead of an external input line as an acquisition trigger. Encoder-triggering is valid only for auto and semi trigger models. (See **ccTriggerProp** on page 2229 and **ccTriggerModel** on page 2221.) This value is ignored if manual triggering is used.

  For platforms that support only single encoder triggering, either the *useXEncoder* or the *useYEncoder* argument can be true, but not both. In these cases, the camera port and encoder port combination will determine which argument must be set true.

Parameters

useXEncoder True to use a single or dual encoder. False to use external triggering or only the y-encoder (which is a second independent encoder).

useYEncoder True to use the y-encoder. False to use external triggering or only the x-encoder.

- `bool encoderTriggerEnabled() const;`

  Returns true if the acquisition uses an encoder instead of an external input line as an acquisition trigger.

xEncoderTriggerEnabled

```
bool xEncoderTriggerEnabled() const;
```

Returns true if the acquisition uses the x-encoder as an acquisition trigger

yEncoderTriggerEnabled

```
bool yEncoderTriggerEnabled() const;
```

Returns true if the acquisition uses the y-encoder as an acquisition trigger

stepsPerLine

```
void stepsPerLine(c_Int32 stepsPerLine,
  c_Int32 step16thsPerLine = 0);

c_Int32 stepsPerLine() const;
```

- `void stepsPerLine(c_Int32 stepsPerLine, c_Int32 step16thsPerLine = 0);`

  Sets the number of encoder steps and fractional steps per line that correspond to a single line of an image. The valid range for *stepsPerLine* depends on the encoder and on the hardware that interfaces with the encoder.

  Parameters

  stepsPerLine The number of encoder steps per line

  step16thsPerLine The number fractional steps per line as 1/16th steps that can be added to the *stepsPerLine* value to make up a single line.

Throws

ccEncoderProp::BadParams

stepsPerLine is not in the range 1 through *maxStepsPerLine*, inclusive. This is the range allowed by the hardware.
step16thsPerLine is not in the range 0 through 15, inclusive.

- `c_Int32 stepsPerLine() const;`

Returns the number of encoder steps that correspond to a single line of an image.

Example

How to calculate the *stepsPerLine* that produce square pixels in the resulting image.

This example uses the following assumptions:

- Encoder resolution is 10 pulses per micrometer or 40 encoder counts per micrometer since one encoder pulse produces four encoder counts.
- The encoder is moving at 10 millimeters per second or 10 micrometers per millisecond.
- The optical magnification is such that 0.5 micrometers in object height maps to the pixel height of the sensor (one image line).

Calculate *stepsPerLine* as follows:

$$stepsPerLine = pixelHeight \times (encoderCounts)/(micrometer) = 0.5 \times 40 = 20$$

So, to obtain square pixels, set *stepsPerLine* = 20 for this example.

Example

Determining maximum speed.

A common enhancement is to speed up an application by increasing the speed of the objects passing under the line scan camera. The objective is to calculate the maximum speed that will still produce square pixels and not cause acquisition failures.

We start with the minimum *stepsPerLine* formula as follows:

$$minStepsPerLine = (encoderCountsPerSecond \times effectiveSensorScanTime)$$

Where *effectiveSensorScanTime* is the greater of the sensor scan time or the exposure time. Sensor scan time and exposure time are covered in the following paragraphs.

Sensor Scan Time

Sensor scan time is the time required to read out the line sensor in the camera. Sensor scan time depends on the sensor width and the camera clock speed. The camera speed is adjustable from 20 MHz to 40 MHz. See **ccDigitalCameraControlProp::masterClockFrequency()**.

The following table shows the minima and maxima of the sensor scan time range. There is a linear relationship between the camera speed settings and the sensor scan time.

Sensor width	Minimum sensor scan time, camera speed = 40 MHz	Maximum sensor scan time, camera speed = 20 MHz
1K pixels	27 microseconds	54 microseconds
2K pixels	54 microseconds	108 microseconds

For a 1K sensor, the sensor scan time can be in the range from 27 to 54 microseconds. For a 2K sensor the range is 54 to 108 microseconds.

Exposure Time

Some line scan camera and platform combinations support adjustable exposure. For these systems exposure time can be set in a range from 0 to a maximum value depending on the camera speed setting as follows:

- 204.8 microseconds at 40 MHz
- 409.6 microseconds at 20 MHz

For line scan camera-platform combinations that do not support adjustable exposure, the exposure time is the time to move the object under the camera one line.

Calculation

For this example we assume an exposure time of 30 microseconds and a 27 microsecond scan time. Therefore, we use an *effectiveSensorScanTime* = 30 microseconds and the assumptions from the previous example in the *minStepsPerLine* formula below. We constrain the *minStepsPerLine* = 20 to ensure square pixels and specify the encoder speed as an unknown:

$$minStepsPerLine = (encoderCountsPerSecond \times effectiveSensorScanTime) + 1$$

$$20 = \left(\left(\frac{40counts}{\mu m} \times \frac{Xmm}{Sec}\right) \times 30\mu Sec\right) + 1$$

Solving for X, X = 16 mm/sec.

So, whereas in the original example assumptions we used a speed of 10 mm/sec, we have shown that for this application the speed could be increased to 16 mm/sec while still maintaining square pixels.

For speeds faster than 16 mm/sec you will get **ccAcqFailure::isTooFastEncoder()** errors.

Test Encoder

When using the test encoder (see **useTestEncoder()** on page 1205), the encoder pulses are applied to the acquisition circuitry at the rate:

encoderCountsPerSecond = ccDigitalCameraProp::masterClockFrequency() * 1e6 / *64*

The minimum steps per line you can use without getting **isTooFastEncoder()** errors is, in the general case:

*minStepsPerLine = encoderCountsPerSecond * effectiveSensorScanTime* + 1

For example, using the table in *Sensor Scan Time* on page 1366, the minimum steps per line for a 1024 pixel wide sensor operating at a 40 MHz master clock frequency is:

(40 * 1e6 / 64) * (27 μs) + 1 = 18

If the test encoder is used for acquiring images of a moving object, the number of steps per line is calculated as:

*stepsPerLine = encoderCountsPerSecond * distancePerLine / speed*

where *distancePerLine* is the physical distance, line to line. The time per line (*distancePerLine / speed)* must be greater than the minimum sensor scan time.

The test encoder can also be used for acquiring images from the test line scan camera (see **cfLSTest_2048()** on page 2169). You can choose to use a portion of the image generated by the test line scan camera, in which case the minimum steps per line you can use without getting **isTooFastEncoder()** errors is:

$$minStepsPerLine = \frac{width + xOffset}{64} + 2$$

where *width* is the width of the region of interest and *xOffset* is the x-offset of the region of interest (see **ccRoiProp** on page 1999). If you are not using an x-offset, the smallest value you can use with the test encoder without getting **isTooFastEncoder()** failures is 34 for 2048 pixel-wide images and 18 for 1024 pixel-wide images.

step16thsPerLine

```
c_Int32 step16thsPerLine() const;
```

Retrieves the number fractional steps per line as 1/16th steps that can be added to the *stepsPerLine* value to make up a single line:

$$EncoderStepsPerLine = stepsPerLine + \frac{step16thsPerLi}{16}$$

Notes

This function is not supported on MVS-8600 and MVS-8600e platforms.

encoderPort

```
void encoderPort(c_Int32 encoderPort);

c_Int32 encoderPort() const;
```

- `void encoderPort(c_Int32 encoderPort);`

Sets the encoder port to use with the FIFO associated with this property. The default value is zero.

Parameters

encoderPort The encoder port.

Notes

This property is used only for encoder-triggered acquires. See **encoderTriggerEnabled()** on page 1207.

On some platforms the association between a camera port and an encoder port is fixed: each encoder can trigger an acquisition to a subset of all the available camera ports on each frame grabber. This means that the encoder port you specify

must be consistent with the camera port of the FIFO that this property is associated with. Unsupported encoder port settings are ignored. See the documentation for the Cognex Video Module (CVM) that you are using for the correspondence between encoder ports and camera ports for your CVM.

Throws

ccEncoderProp::BadParams

i is not in the range zero to **numEncoderPort()** - 1.

- `c_Int32 encoderPort() const;`

Returns the encoder port used with the FIFO associated with this property.

numEncoderPort

`c_Int32 numEncoderPort() const;`

Returns the number of encoder ports that the frame grabber used by the FIFO associated with this property.

Throws

ccEncoderProp::NoFrameGrabber

This property is not associated with a FIFO.

encoderOffset

`void encoderOffset(c_Int32 encoderOffset);`

`c_Int32 encoderOffset() const;`

- `void encoderOffset(c_Int32 encoderOffset);`

Sets the encoder position at which line zero of the region of interest occurs.

The *encoderOffset* value is only part of the calculation of the encoder position where acquisition will start when using encoder-triggered acquisition. The parameters included in the calculation are the **ccRoiProp::roi().ul().y()** line number, the steps per line, and this offset. For positive direction acquisitions, the starting encoder count is:

```
// For this example, y0 means ccRoiProp::roi().ul().y()
startCount = (y0 * stepsPerLine) + encoderOffset.
```

For negative direction acquires the starting encoder count is:

```
startCount = (-y0 * stepsPerLine) + encoderOffset.
```

Region of interest values that exceed the maximum width, height, or size of the camera format result in a **ccAcqFailure::isInvalidRoi()** error.

Parameters

encoderOffset The ROI offset in encoder counts.

- `c_Int32 encoderOffset() const;`

  Gets the encoder position at which line zero of the region of interest occurs.

positiveAcquireDirection

```
void positiveAcquireDirection(bool isPositive);

bool positiveAcquireDirection() const;
```

- `void positiveAcquireDirection(bool isPositive);`

  Specifies whether encoder values increase or decrease during image acquisition. The default value is true.

  Parameters

  isPositive If true, encoder values increase during image acquisition; if false, encoder values decrease during image acquisition.

- `bool positiveAcquireDirection() const;`

  Returns true if encoder values increase during image acquisition; false if encoder values decrease during image acquisition.

encoderTriggerLatency

```
c_Int32 encoderTriggerLatency() const;
```

Returns the latency between the encoder offset to trigger acquisitions and the start of the acquisition in encoder counts based on the most recent encoder-triggered acquisition. If this property is not associated with a FIFO, this function returns zero.

useSingleChannel

```
void useSingleChannel(bool isSingleChannel);

bool useSingleChannel() const;
```

Specifies whether only one channel, or two channels of the encoder signal are used for acquisition.

- `void useSingleChannel(bool isSingleChannel);`

  Set to *true* to use a single channel. Set to *false* to use both channels.

  Parameters
  isSingleChannel True or false.

- `bool useSingleChannel() const;`

  Returns true if set for single channel. Returns false if set for two channels.

  Notes
  The default value is false indicating both encoder channels A and B are used for acquisition on platforms supporting this functionality.

  If **useSingleChannel()** is set to true, it might be possible to connect only a single encoder channel as the encoder input instead of using a quadrature dual channel type encoder. The channel that is used depends upon the setting of **positiveAcquireDirection()**. If **positiveAcquireDirection()** is true, channel A is used. If false, channel B is used.

  An advantage of using a single channel encoder input is that in some cases it allows you to acquire without considering the motion direction of the encoder.

  The MVS-8600 and MVS-8600e are the only platforms that support this function.

encoderResolution

```
void encoderResolution(ceEncoderResolution encRes);
ceEncoderResolution encoderResolution() const;
```

Specifies the encoder resolution.

- `void encoderResolution(ceEncoderResolution encRes);`

  Sets the encoder resolution. Must be one of the **ceEncoderResolution** enums.

  Parameters
  encRes Encoder resolution.

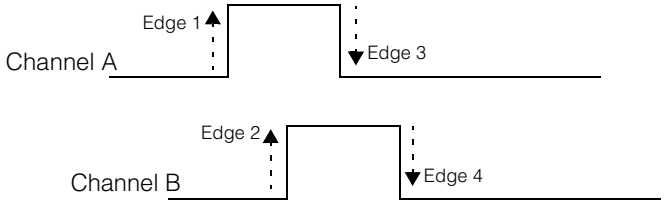
- `ceEncoderResolution encoderResolution() const;`

  Returns the currently specified encoder resolution.

Notes

The default value is *e1xEncoderResolution* indicating that the encoder pulse from a full clock cycle produces one count.

On a quadrature encoder input, by counting half or all transition edges on both channel A and channel B it is possible to get either 2x or 4x encoder input resolution. See the following diagram.



e1xEncoderResolution = 1 count from edge 1
e2xEncoderResolution = 2 counts from edges 1 and 2
e4xEncoderResolution = 4 counts from edges 1, 2, 3 and 4

The 4x encoder resolution setting is recommended, if available, and is the only setting available on the MVS-8120 with a CVM11.

For historical reasons, the MVS-8600 and MVS-8600e default to 1x.

The 4x setting with *StepsPerLine* = 4, is equivalent to the 1x setting with *StepsPerLine* = 1.

The 4x setting with *StepsPerLine* = 2 (acquires every other edge) is not equivalent to the 2x setting with *StepsPerLine* = 1 (acquires for two edges, skips two edges). See non-doc comment above.

The MVS-8600e is the only platform that fully implements this function. Refer to the MVS-8600 and MVS-8600e Hardware Manual for more information.

startAcqOnEncoderCount

```
void startAcqOnEncoderCount(c_Int32 encoderCount);

c_Int32 startAcqOnEncoderCount() const;
```

Specifies the number of encoder counts the hardware waits before beginning an acquisition. The default value is 0 which means the acquisition starts without waiting for any encoder counts. The encoder count value can be any value in the range 0 through 65535.

Also, If there are any stored backward counts, new acquisitions will not begin until the backward count is compensated by forward encoder counts.

- `void startAcqOnEncoderCount(c_Int32 encoderCount);`

  Sets the encoder count.

  Parameters

  encoderCount The number of encoder counts to wait.

- `c_Int32 startAcqOnEncoderCount() const;`

  Returns the current encoder count setting.

  Notes

  The *encoderCount* value affects acquisitions using all trigger models.

  When using trigger models that require a hardware trigger to start an acquisition, the acquisition will only start when the hardware trigger occurs and after *encoderCount* encoder counts are generated.

  For MVS-8600e frame grabbers there is no absolute encoder position counter available. There is however, a counter that stores the count of any backward encoder pulses detected by the hardware when the encoder is moving in the opposite direction to the one expected for acquisition. If there are any stored backward counts, new acquisitions will not begin until the backward motion is compensated by forward motion. Setting a non-default *encoderCount* value using this function will set the backward encoder pulse counter to this value.

  The MVS-8600e is the only platform that fully implements this function.

ignoreBackwardEncoderCountsBetweenAcquires

```
void ignoreBackwardEncoderCountsBetweenAcquires(
                                                    bool ignore);

bool ignoreBackwardEncoderCountsBetweenAcquires() const;
```

This function allows you to ignore the backward encoder counts described in **startAcqOnEncoderCount** above. When set *true*, backward encoder counts are ignored. When set false backward encoder counts are treated as described above.

- `void ignoreBackwardEncoderCountsBetweenAcquires(`
  `bool ignore);`

  Parameters

  ignore Sets the ignore flag to this value; true or false.

- `bool ignoreBackwardEncoderCountsBetweenAcquires() const;`

  Returns the current state of the *ignore* flag.

  Notes

  The default value is false which means that the backward encoder motion is tracked between acquisitions. This setting can be useful if there is unexpected backward motion between acquires and acquisition should not start until that unexpected backward motion is compensated.

  Setting a value of true will make the hardware ignore any backward encoder motion between acquisitions. This setting can be useful if between acquisitions there is deliberate backward motion that you wish to ignore when the next acquisition starts.

  The MVS-8600e is the only platform that supports this function.

ignoreTooFastEncoder

```
void ignoreTooFastEncoder(bool ignore);

bool ignoreTooFastEncoder() const;
```

Specifies whether or not to ignore the acquisition failure *isTooFastEncoder*. See **ccAcqFailure::isTooFastEncoder()**.

- `void ignoreTooFastEncoder(bool ignore);`

  Sets the ignore flag to true or false.

Parameters

ignore The ignore flag, true or false.

- `bool ignoreTooFastEncoder() const;`

  Returns the current value of the ignore flag, true or false.

  Notes

  The default value is true which means line overrun conditions will not generate an *isTooFastEncoder* acquisition failure. This setting may be useful in cases where an occasional line overrun condition can occur and the application can still handle the image.

  Setting a value of false will make the hardware report an *IsTooFastEncoder* acquisition failure when a line overrun condition occurs. See **ccAcqFailure::isTooFastEncoder()** for more information about this error.

  The MVS-8600e is the only platform that supports this function.

Constants

defaultStepsPerLine

```
static const c_Int32 defaultStepsPerLine;
```

The default steps per line value: 3.

maxStepsPerLine

```
static const c_Int32 maxStepsPerLine;
```

The maximum steps per line: 4095.

defaultStep16thsPerLine

```
static const c_Int32 defaultStep16thsPerLine;
```

The default fractional steps per line value: 0.

defaultEncoderResolution

```
static const ceEncoderResolution defaultEncoderResolution;
```

The default value is *e1xEncoderResolution*.

defaultStartAcqOnEncoderCount

```
static const c_Int32 defaultStartAcqOnEncoderCount;
```

The default value is 0.

ccEvent

```
#include <ch_cvl/threads.h>

class ccEvent: public cc_Resource;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A class that describes an event. An event can have two states, set and reset. A thread can change the state of an event, and it can block waiting for an event's state to change from reset to set.

Constructors/Destructors

ccEvent

```
ccEvent(bool manualReset=true,
  bool initiallySignaled=false,
  ccCvlString name = cmT(""));
```

Construct a **ccEvent** with the specified name, state, and auto-reset behavior.

All **ccEvent**s constructed with the same name refer to the same global event.

Parameters

manualReset — If true, whenever any thread changes the state of this **ccEvent** to set, then all threads that had locked this **ccEvent** are unblocked. If *manualReset* is false, then only the first thread that locked this **ccEvent** is unblocked and the **ccEvent** is automatically reset to the reset state.

initiallySignaled — If true, this **ccEvent** is constructed in the set state. If false, this **ccEvent** is constructed in the reset state.

name — The name of this **ccEvent**. Whenever any thread constructs a **ccEvent**, if another **ccEvent** exists with the same name, the two **ccEvent**s refer to the same global event.

If you specify a zero-length string for *name* (the default value for the argument), an unnamed **ccEvent** is constructed. Unnamed events are always separate instances.

Throws

cc_Resource::NamedObjectExists
: An underlying system mutex or semaphore with the specified name already exists.

setEvent

```
bool setEvent();
```

Changes the state of this **ccEvent** to set. If *manualReset* was set to true on construction, then all threads waiting for this event are unlocked until **resetEvent()** is called. If *manualReset* was set to false on construction, then the first thread that locked this event is unlocked and this **ccEvent**'s state is changed to reset.

This function returns true if the operation was successful, false if it failed.

resetEvent

```
bool resetEvent();
```

Changes the state of the event to reset.

This function returns true if the operation was successful, false if it failed.

lock

```
bool lock(double timeout=HUGE_VAL);
```

The current thread is blocked until this **ccEvent**'s state is becomes set. This function returns true if this **ccEvent**'s state became set, false if the timeout expired.

Parameters

timeout
: The number of seconds to block. If you specify *HUGE_VAL* for *timeout*, the thread will wait forever.

Throws

cc_Resource::BrokenLock
: The **cc_Resource::breakLocks()** static function was invoked on this thread.

unlock

```
void unlock();
```

Changes the state of the event to set. If *manualReset* was set to true on construction, then all threads waiting for this event are unlocked. If *manualReset* was set to false on construction, then the first thread that locked this event is unlocked and this **ccEvent**'s state is changed to reset. This function has the same effect as **setEvent()**.

ccException

```
#include <ch_cvl/except.h>

class ccException: public ccPersistent;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

This is a base class from which are derived all CVL exceptions.

Constructors/Destructors

You cannot construct a **ccException** directly.

Public Member Functions

message

```
ccCvlString message(bool includeClass=true) const;

void message( ccCvlOStream& str) const;
```

- `ccCvlString message(bool includeClass=true) const;`

  Returns a message describing this exception.

  If *includeClass* is true, the message consists of a type name, followed by a colon, followed by the results of a call to **message_()**. Otherwise, the message consists of the results of a call to **message_()**.

  Parameters

  includeClass — If true, the message returned by this function is prepended with the type name and a colon.

- `void message( ccCvlOStream& str) const;`

  Prints a message describing this exception to the supplied output stream. This function calls the first overload of **message()** to obtain the string.

  Parameters

  str — The stream to which to print the message.

errorNumber

```
virtual c_Int32 errorNumber() const=0;
```

This function returns a unique 32-bit value associated with this exception. The error numbers for CVL exceptions are contained in the files in *ch_err*.

You must override this function in any non-abstract class you derive from **ccException**. This function is called by the exception handling mechanism to obtain the unique 32-bit error number associated with an exception.

Cognex-supplied exceptions use values of 0x10000000 and greater; your override of this function should return a unique value in the range 0x00000001 through 0x0FFFFFFF. Values less than 0x00000001 are reserved. A value of 0x00000000 can be used to indicate “no error”.

Notes

Overrides of this member function should return a unique error number for each derived class. No mechanism is provided to automatically ensure uniqueness.

Protected Member Functions

message_

```
virtual void message_( ccCvlOStream& str) const;
```

This function is called by the exception handling mechanism to obtain a human-readable error string. You should override this function in each class that you derive from **ccException**.

The implementation of **message_()** in **ccException** prints "no further information provided" (with no newline character).

Parameters

str	The stream to which to print the message

Related Classes

```
class ccBadPointer : public ccException;

class ccBadUnicodeChar : public ccException;

class ccInvariantFailure : public ccException;

class ccUnsupportedHW : public ccException;

class ccSecurityViolation : public ccExceptionWithString;
```

These global exceptions are derived directly from **ccException**. In most cases, the causes of these exceptions are self-explanatory. It is unlikely that you will want to catch these exceptions explicitly.

- `class ccBadPointer : public ccException;`

  An attempt was made to reference memory using an invalid pointer.

- `class ccBadUnicodeChar : public ccException;`

  An invalid Unicode character was encountered.

- `class ccInvariantFailure : public ccException;`

  An internal consistency check failed; the state of some CVL object is invalid.

- `class ccUnsupportedHW : public ccException;`

  A call was made to a software tool or subsystem which does not work with the Cognex hardware device installed in this system, or with the host computer system's CPU.

- `class ccSecurityViolation : public ccExceptionWithString;`

  A license key was not available. Contact Cognex Technical Support for assistance.

ccException

ccExceptionWithString

```
#include <ch_cvl/except.h>

class ccExceptionWithString: public ccException;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

This is a base class from which you can derive exceptions for which you can specify multiple messages. Exceptions which are derived from **ccException** must have their messages defined in their override of **ccException::message_()**.

For information on how to derive classes from **ccExceptionWithString**, consult the chapter *Exception Handling* in the *CVL Users Guide*.

Constructors/Destructors

You cannot construct a **ccExceptionWithString** directly.

Protected Member Functions

ccExceptionWithString

```
ccExceptionWithString();

ccExceptionWithString(const ccCvlString& info);

ccExceptionWithString(const TCHAR* info);
```

You can only call these constructors as part of the construction of an object derived from **ccExceptionWithString**. The supplied string will be returned by calls to the derived object's **message()** function.

- `ccExceptionWithString();`

  Constructs a **ccExceptionWithString** that returns "no further information provided".

ccExceptionWithString

- `ccExceptionWithString(const ccCvlString& info);`

  Constructs a **ccExceptionWithString** that returns the supplied string as its message.

  Parameters

  info The message to return.

- `ccExceptionWithString(const TCHAR* info);`

  Constructs a **ccExceptionWithString** that returns the supplied string as its message.

  Parameters

  info The message to return.

ccExposureProp

```
#include <ch_cvl/prop.h>

class ccExposureProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class allows you to set the exposure property of an acquisition FIFO. The exposure property is used for shuttered and strobed acquisitions. For shuttered acquisitions, the exposure property specifies how long the camera shutter is open. For strobed acquisitions, you use the exposure property to report the duration of the strobe flash to the acquisition software.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccExposureProp

```
ccExposureProp();

explicit ccExposureProp(double);
```

- `ccExposureProp();`

  Creates a new exposure property not associated with any FIFO. The exposure duration is set to *ccExposureProp::defaultExposure*.

- `explicit ccExposureProp(double seconds);`

  Set the exposure duration to *seconds*. For shuttered acquisitions, this value is the amount of time to leave the shutter open. For strobed acquisitions, this value is the duration of the strobe flash.

Parameters

seconds The exposure duration in seconds. A value of zero means to use the shortest supported exposure time.

Throws

ccExposureProp::BadParams
seconds was less than zero.

Public Member Functions

exposure

```
void exposure(double seconds);
double exposure();
```

- `void exposure(double seconds);`

For shuttered acquisitions, sets the exposure duration to *seconds*.

For strobed acquisitions, this value is the duration of the strobe flash. To find the flash duration of your strobe, see the strobe light's specifications. Note that this function is used to let the acquisition software know the duration of the strobe flash, not to set the duration of the strobe flash.

Parameters

seconds The exposure duration in seconds. A value of zero means to use the shortest supported exposure time.

Notes

On most platforms, *seconds* has a granularity of 1 millisecond to 1 microsecond.

Throws

ccExposureProp::BadParams
seconds was less than zero.

- `double exposure() const;`

Return the exposure time in seconds.

Constants

defaultExposure `static const double defaultExposure;`

The default exposure value: 20e–6. This value is appropriate for strobed acquisition. If you use ambient lighting with a shuttered camera such as the Sony XC-55, you may need to increase the exposure time.

ccFeaturelet

```
#include <ch_cvl/feature.h>

class ccFeaturelet
```

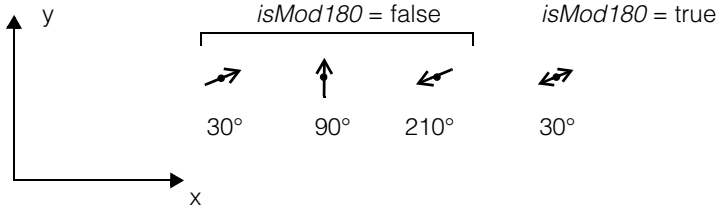
Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

The Featurelet class characterizes features which have optional weights and magnitudes. Each featurelet has the following characteristics:

Characteristic	Default value	Description
position	(0, 0)	The x,y location of the featurelet
angle	0°	The featurelet orientation
weight	1	An importance factor you can assign
magnitude	0	A relative size factor you can assign
isMod180	false	When true, the orientation is only valid mod 180° rather than the normal mod 360°

Featurelets are typically shown as a dot indicating the position, with an overlaid arrow to indicate the featurelet orientation.



The *position* is the point in x,y space where the featurelet is located. The featurelet angle is shown by the orientation of the arrow. A featurelet with an arrowhead on both ends indicates that isMod180 = true. This means that if you rotate the featurelet 180° its description does not change.

Constructors/Destructors

ccFeaturelet

```
explicit ccFeaturelet(
  const cc2Vect &position = cc2Vect(),
  const ccRadian &angle = ccRadian(0),
  double Magnitude = 0,
  double weight = 1,
  bool isMod180 = false);
```

Constructor; sets the parameters to the specified values.

Parameters

position	The (x,y) position of the featurelet.
angle	The featurelet angle, in radians.
Magnitude	The featurelet magnitude.
weight	The featurelet weight.
isMod180	True if the featurelet angle is mod 180°. False otherwise.

Throws

ccFeatureletDefs::BadParams
If *magnitude* < 0.

Operators

operator==

```
bool operator==(const ccFeaturelet& rhs) const;
```

Comparison operator. Compares this **ccFeaturelet** object to another **ccFeaturelet** object. Returns true if they are equal, and false if they are not equal.

Two **ccFeaturelet** objects are considered equal if, and only if, all their members are exactly equal.

Parameters

rhs	The **ccFeaturelet** object to compare to this one.

Public Member Functions

position

```
const cc2Vect &position() const;
void position(const cc2Vect &position);
```

- `const cc2Vect &position() const;`

  Returns the featurelet (x,y) position.

- `void position(const cc2Vect &position);`

  Sets a new featurelet (x,y) position.

  Parameters

  position The new position.

angle

```
ccRadian angle() const;
void angle(ccRadian angle);
```

The featurelet angle expressed in radians.

- `ccRadian angle() const;`

  Returns the featurelet angle.

- `void angle(ccRadian angle);`

  Sets the featurelet *angle* to the specified number of radians.

  Parameters

  angle The new angle in radians.

magnitude

```
double magnitude() const;
void magnitude(double magnitude);
```

A featurelet can be assigned a magnitude value expressed as zero or a positive number.

- `double magnitude() const;`

  Returns the magnitude value.

- `void magnitude(double magnitude);`

  Sets a new magnitude value.

  Parameters
  magnitude The new value.

  Throws
  ccFeatureletDefs::BadParams
  If *magnitude* < 0.

weight

```
double weight() const;
void weight(double weight);
```

A featurelet can be assigned a weight expressed as double.

- `double weight() const;`

  Returns the currently assigned weight.

- `void weight(double weight);`

  Sets a new weight.

  Parameters
  weight The new weight.

isMod180

```
bool isMod180() const;
void isMod180(bool isMod180);
```

If the featurelet angle is mod 180° it means the orientation information is only valid mod 180° rather than the normal mod 360°. Featurelet orientation normally indicates the gradient direction from dark to light. When a featurelet has **isMod180()** set true it generally means the gradient polarity is not known or cannot be determined.

- `bool isMod180() const;`

  Returns true if the featurelet angle is mod 180°. Returns false otherwise.

- `void isMod180(bool isMod180);`

  Sets the featurelet mod 180 state, true or false.

Parameters

isMod180 The new mod 180° state.

map

```
ccFeaturelet map(
  const cc2XformBase &xform) const;
```

Return a new featurelet corresponding to this featurelet mapped by the given transform.

Parameters

xform The mapping transform.

Notes

The new featurelet's *position* and *angle* are the original featurelet's *position* and *angle* mapped by the linearized transform at the original featurelet's position. The new featurelet's *magnitude*, *weight*, and *isMod180* are equal to the original featurelet's *magnitude*, *weight*, and *isMod180*.

This function maps angles in a non intuitive way. For example, the computed angle is not simply **xform.mapAngle(angle)**. If you want the featurelet angle computed as **xform.mapAngle(angle)**, you must write your own map function to do this.

The featurelet angle corresponds to the gradient which is normal to the featurelet tangent. The function maps the featurelet angle by first subtracting 90° to obtain the tangent angle. Then, mapping the tangent angle and adding the 90° back to the mapped angle. (For a handedness flip, 90° is subtracted from the mapped angle instead of added to the mapped angle).

ccFeaturelet

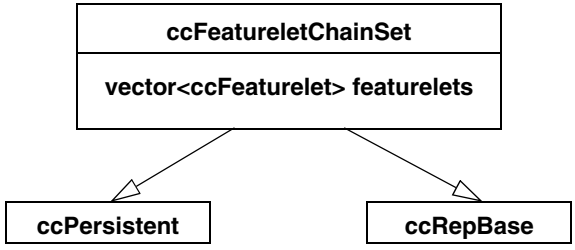
ccFeatureletChainSet

```
#include <ch_cvl/feature.h>

class ccFeatureletChainSet : public virtual ccPersistent,
                             public virtual ccRepBase
```

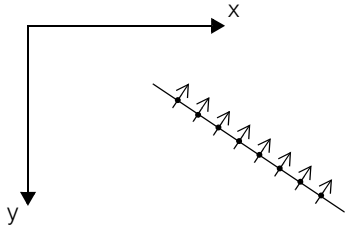
Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex



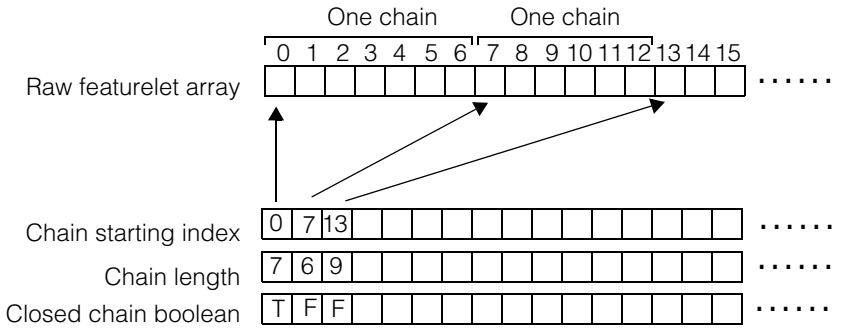
The Featurelet Chain Set class characterizes a set of chains of featurelets. (See the **ccFeaturelet** class reference page for a description of featurelets). There is an abstract notion of a single featurelet chain, but there is no class characterizing a single featurelet chain. A featurelet chain can be either open or closed. Note that open and closed featurelet chains can have any number of featurelets, including zero. There is no limit on the number of featurelets in any chain.

The example below shows a chain of featurelets along a line.



The Featurelet Chain Set class includes a vector of raw featurelets. Each chain in the set corresponds to a contiguous subset of these raw featurelets. The featurelet chain set also includes a vector of chain start indices and a vector of chain lengths. These vectors

combine to specify the contiguous featurelet subsets corresponding to each chain. The featurelet chain set also includes a vector of booleans corresponding to whether each chain is open or closed. See the following example diagram.



Note that every featurelet in the raw vector is an element of at most one chain; that is, there are no shared featurelets in the raw vector.

However, it is legal for a featurelet to belong to no chain. The most typical use cases are the following:

1. A set of featurelets where every featurelet belongs to exactly one chain, that is, there are no unchained featurelets.
2. A set of featurelets with zero chains, that is, no featurelets are chained.

However, in the most general case, some featurelets might belong to chains while other featurelets might not belong to any chain. Users who wish to iterate through all featurelets regardless of whether or not those featurelets are chained should use the featurelets() vector getter; users who wish to iterate through only chained featurelets should use the featureletChains() vector getter.

All featurelet chains are composed of a contiguous set of featurelets in the raw featurelet array.

Constructors/Destructors

ccFeatureletChainSet

```
ccFeatureletChainSet();

ccFeatureletChainSet (
  const cmStd vector<cmStd vector<ccFeaturelet>>
&chainedFeaturelets,
```

```
  const cmStd vector<bool> &isClosed,
  const cmStd vector<ccFeaturelet> &unchainedFeaturelets =
cmStd vector<ccFeaturelet>());

ccFeatureletChainSet(
  const ccFeatureletChainSet &chainSet);

virtual ~ccFeatureletChainSet();
```

- `ccFeatureletChainSet();`

  Default constructor. Constructs a featurelet chain set with no chains.

- ```
  ccFeatureletChainSet (
    const cmStd vector<cmStd vector<ccFeaturelet>>
  &chainedFeaturelets,
    const cmStd vector<bool> &isClosed,
    const cmStd vector<ccFeaturelet> &unchainedFeaturelets =
  cmStd vector<ccFeaturelet>());
  ```

  Constructor. Initializes the chain set with the given chained featurelets and *isClosed* vectors, and optionally also with the given unchained featurelets.

  Parameters

  chainedFeaturelets
  A vector of vectors holding chained featurelets. Each interior vector is a chain of featurelets. The next level vector is a vector of chains, a chain set.

  isClosed A corresponding vector of booleans, one for each chain.

  unchainedFeaturelets
  A vector holding unchained featurelets.

  Throws

  ccFeatureletDefs::BadParams
  If **chainedFeaturelets.size()** != **isClosed.size()**

- ```
  ccFeatureletChainSet(
    const ccFeatureletChainSet &chainSet);
  ```

  Copy constructor; makes a deep copy.

- `virtual ~ccFeatureletChainSet();`

  Destructor.

Operators

operator=

```
ccFeatureletChainSet& operator=(
  const ccFeatureletChainSet &chainSet);
```

Assignment operator; makes deep copy.

operator==

```
bool operator==(const ccFeatureletChainSet& rhs) const;
```

Comparison operator. Returns true if *rhs* is equal to this object. Returns false otherwise.

Two **ccFeatureletChainSet** objects are considered equal if and only if all their members are exactly equal.

Parameters

rhs The comparison object.

Public Member Functions

reset

```
void reset();
```

Reset this featurelet chain set object to its default constructed state.

replaceChain

```
void replaceChain (
  c_Int32 chainIndex,
  const cmStd vector<ccFeaturelet> &featurelets,
  bool isClosed);
```

Replace a chain in this object with the chain provided.

This may require repositioning featurelets in raw featurelet array with chain indices > *chainIndex*.

Parameters

chainIndex The index of the chain in this object that is to be replaced.

featurelets The new chain of featurelets.

isClosed True if the new chain is closed, false otherwise.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

Notes

The initial featurelet chain set remains unchanged if there is a throw.

insertChain

```
void insertChain (
  c_Int32 chainIndex,
  const cmStd vector<ccFeaturelet> &featurelets,
  bool isClosed);
```

Insert the given chain into the chain set contained in this object. The chain is inserted at index position *chainIndex*. which becomes its index in the modified chain. Indices for all chains after the inserted chain will be incremented by 1.

This will require repositioning featurelets in raw featurelet array with chain indices > *chainIndex*.

Parameters

chainIndex	The index where that new chain is inserted.
featurelets	The new chain.
isClosed	True if the new chain is closed, false otherwise.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* > **numChains()**.

Notes

The initial featurelet chain set remains unchanged if there is a throw.

addChain

```
void addChain (
  const cmStd vector<ccFeaturelet> &featurelets,
  bool isClosed);
```

Add a chain to the chain set contained in this object. The chain is appended and becomes the last chain.

Parameters

featurelets	The added chain.
isClosed	True if the new chain is closed, false otherwise.

deleteChain

```
void deleteChain(c_Int32 chainIndex);
```

Delete the specified chain from the chain set contained is this object.

This will require repositioning featurelets in raw featurelet array with chain indices > *chainIndex*.

Parameters

chainIndex The index of the chain to be deleted.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

Notes

The initial featurelet chain set remains unchanged if there is a throw.

append

```
void append(
  const ccFeatureletChainSet &featureletChainSet);
```

Append the provided featurelet chain set to the chain set contained in this object.

Parameters

featureletChainSet
The chain set to append.

numChains

```
c_Int32 numChains() const;
```

Returns the number of chains in the chain set contained in this object.

Notes

The featurelet chain set may include unchained featurelets, and thus even a chain set with zero numChains() may still contain featurelets. Use the featurelets() vector getter to access the featurelets.

numFeaturelets

```
c_Int32 numFeaturelets() const;
```

Returns the number of featurelets in the raw featurelet array contained in his object, regardless of whether a featurelet is contained by a chain or not.

reserve

```
void reserve(
  c_Int32 numChains,
  c_Int32 numFeaturelets) const;
```

Reserve the specified number of chains and featurelets for this chain set. This reserves array space for chains you anticipate adding.

Parameters

numChains The number of chains to reserve.

numFeaturelets The number of featurelets to reserve.

Throws
ccFeatureletDefs::BadParams
If *numChains* < 0,
or if *numFeaturelets* < 0.

isClosed

```
bool isClosed(c_Int32 chainIndex) const;
cmStd vector<bool> isClosed() const;
```

- `bool isClosed(c_Int32 chainIndex) const;`

Returns true if the chain specified by *chainIndex* is closed. Returns false otherwise.

Parameters
chainIndex The index of the chain to query.

Throws
ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

- `cmStd vector<bool> isClosed() const;`

Returns the vector of booleans that indicate which chains are closed and which are open. See the introductory description and the diagram.

chainLength

```
c_Int32 chainLength(c_Int32 chainIndex) const;
```

Returns the number of featurelets in the chain specified by *chainIndex*.

Parameters
chainIndex The index of the chain to query.

Throws
ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

startFeatureIndex

```
c_Int32 startFeatureIndex(c_Int32 chainIndex) const;
```

Return the index into the raw featurelet array of the start of the chain specified by *chainIndex*.

Parameters

chainIndex The index of the chain to query.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

---

featurelets

```
const cmStd vector<ccFeaturelet> &featurelets() const;

cmStd vector<ccFeaturelet> featurelets(c_Int32 chainIndex)
const;
```

- `const cmStd vector<ccFeaturelet> &featurelets() const;`

Returns the vector of all featurelets contained in this chain set, regardless of whether or not those featurelets are chained.

Notes

The returned const reference becomes invalid if and when the featurelet chain set is modified. Use the returned vector immediately following this call and then do not use it again.

- `cmStd vector<ccFeaturelet> featurelets(`
  `c_Int32 chainIndex) const;`

Returns the array of featurelets for the chain specified by *chainIndex*.

Parameters

chainIndex The index of the chain to query.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

featureletChains

```
cmStd vector<cmStd vector<ccFeaturelet>>
                                    featureletChains() const;
```

Returns the entire raw featurelets array contained in this object. These are all of the featurelets for all the chains, unchained featurelets are not included.

Notes

If you wish to iterate through all of the featurelets, regardless of whether or not those featurelets are chained, you should use the featurelets() vector getter instead.

featurelet

```
const ccFeaturelet &featurelet(
  c_Int32 chainIndex,
  c_Int32 featureletIndex) const;
```

Returns a reference to a specific featurelet. *chainIndex* specifies the chain and *featureletIndex* specifies the featurelet within the chain.

Notes

The returned const reference becomes invalid if and when the featurelet chain set is modified. Use the returned reference immediately following this call and then do not use it again.

Parameters

chainIndex The specified chain.

featureletIndex The index of the featurelet within the chain.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**,
or if *featureletIndex* < 0,
or if *featureletIndex* >= **chainLength(***chainIndex***)**.

map

```
ccFeatureletChainSetPtrh map(
  const cc2XformBase &xform) const;
```

Return a new featurelet chain set corresponding to the featurelet chain set contained in this object, mapped by the given transform.

Parameters

xform The mapping transform.

Notes

Each new featurelet's position and angle are the original featurelet's position and angle mapped by the linearized transform at each original featurelet's position. The new featurelet's *magnitude*, *weight*, and *isMod180* is equal to the original featurelet's *magnitude*, *weight*, and *isMod180*.

This function maps angles in a non intuitive way. For example, the computed angle is not simply **xform.mapAngle(angle)**. If you want the featurelet angle computed as **xform.mapAngle(angle)**, you must write your own map function to do this.

The featurelet angle corresponds to the gradient which is normal to the featurelet tangent. The function maps the featurelet angle by first subtracting 90° to obtain the tangent angle. Then, mapping the tangent angle and adding the 90° back to the mapped angle. (For a handedness flip, 90° is subtracted from the mapped angle instead of added to the mapped angle).

boundingBox

```
ccRect boundingBox(c_Int32 chainIndex) const;
ccRect boundingBox() const;
```

- `ccRect boundingBox(c_Int32 chainIndex) const;`

  Return an enclosing rectangle of the chain specified by *chainIndex*.

  Parameters

  chainIndex The index of the target chain.

  Throws

  ccFeatureletDefs::BadIndex
  If *chainIndex* < 0,
  or if *chainIndex* >= **numChains()**.

  ccFeatureletDefs::EmptyChain
  If the specified chain is empty.

- `ccRect boundingBox() const;`

  Return the enclosing rectangle of all the chains contained in this object.

  Throws

  ccFeatureletDefs::EmptyChainSet
  If the chain set is empty.

isRightHanded

```
bool isRightHanded(c_Int32 chainIndex) const;
```

Returns true if the specified chain is right-handed. Returns false if it is not. The chain must be closed.

A chain is right-handed when its signed area corresponding to the region specified by the featurelet positions is 0 or positive. The signed area is defined as the sum of all the cross products between successive featurelet positions.

Notes

A featurelet chain with a signed area of zero is defined to be right-handed.

A chain with 0, 1, or 2 featurelets is defined to be right-handed.

Parameters

chainIndex The specified chain.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

ccFeatureletDefs::NotClosedChain
If the chain is not closed.

reverseFeatureletOrder

```
void reverseFeatureletOrder(c_Int32 chainIndex);

void reverseFeatureletOrder();
```

- `void reverseFeatureletOrder(c_Int32 chainIndex);`

Reverse the order of the featurelets in the specified chain.

This has the effect of toggling the chain's handedness unless the signed area is exactly zero or unless the chain length is 0, 1, or 2, in which case the chain remains right-handed.

Reversing the order of the featurelets does not affect whether the chain is open or closed.

Parameters

chainIndex The specified chain.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

Notes

The initial featurelet chain set remains unchanged if there is a throw.

- `void reverseFeatureletOrder();`

Reverse the order of the featurelets in each chain and reverse the order of all unchained featurelets. The chains themselves remain in the same order. This has the effect of toggling the chain's handedness unless the signed area is exactly zero or unless the chain length is 0, 1, or 2, in which case the chain remains right-handed.

Reversing the order of the featurelets does not affect whether a chain is open or closed.

reverseFeatureletOrientations

```
void reverseFeatureletOrientations(c_Int32 chainIndex);
void reverseFeatureletOrientations();
```

- `void reverseFeatureletOrientations(c_Int32 chainIndex);`

  Reverse the angle of all featurelets in the specified chain by adding 180° to each featurelet angle. The new featurelet angles are computed mod 360°.

  Parameters
  chainIndex The specified chain.

  Throws
  ccFeatureletDefs::BadIndex
  If *chainIndex* < 0,
  or if *chainIndex* >= **numChains()**.

  Notes
  The initial featurelet chain set remains unchanged if there is a throw.

- `void reverseFeatureletOrientations();`

  Reverse the angle of all featurelets in the chain set by adding 180° to each featurelet angle. The new featurelet angles are computed mod 360°.

anyFeatureletIsMod180

```
bool anyFeatureletIsMod180(c_Int32 chainIndex) const;
bool anyFeatureletIsMod180() const;
```

- `bool anyFeatureletIsMod180(c_Int32 chainIndex) const;`

  Returns true if any featurelet in the specified chain has a mod 180 orientation. Returns false otherwise.

  Parameters
  chainIndex The specified chain.

  Throws
  ccFeatureletDefs::BadIndex
  If *chainIndex* < 0,
  or if *chainIndex* >= **numChains()**.

- `bool anyFeatureletIsMod180() const;`

  Returns true if any featurelet in this chain set has a mod 180 orientation. Returns false otherwise.

proportionPositiveCrossProduct

```
double proportionPositiveCrossProduct(
  c_Int32 chainIndex) const;
```

Returns the proportion of the featurelets in the specified chain that have positive cross products between both incident vectors (the vectors between successive featurelets) and the featurelet angles (assumed to be the gradient).

This is a measure of the common polarity of the featurelets in the chain. For example, in a circle if all featurelets point inwards towards the circle center, this function will return either 0 or 1.0 depending on whether the featurelet chain navigates the circle boundary in the clockwise, or counter-clockwise direction. If half of the featurelets pointed inwards, and the other half pointer outwards, the function would return 0.5. The returned result is always in the range 0 through 1.0.

For more detail see the *Featurelet Chain Polarity* section below.

Notes

This function assumes that the featurelet angles correspond to gradient orientations from negative to positive.

The proportion computation ignores any featurelets which have mod180 orientation (**isMod180()** == true).

If all of the featurelets have mod180 orientation or there are no featurelets, this function returns exactly 0.5.

When the computed value is 0, 0.5, or 1.0 the function returns exactly 0, 0.5, or 1.0 (not an approximately close floating point number). This is done so you can easily explicitly check for these cases.

Parameters

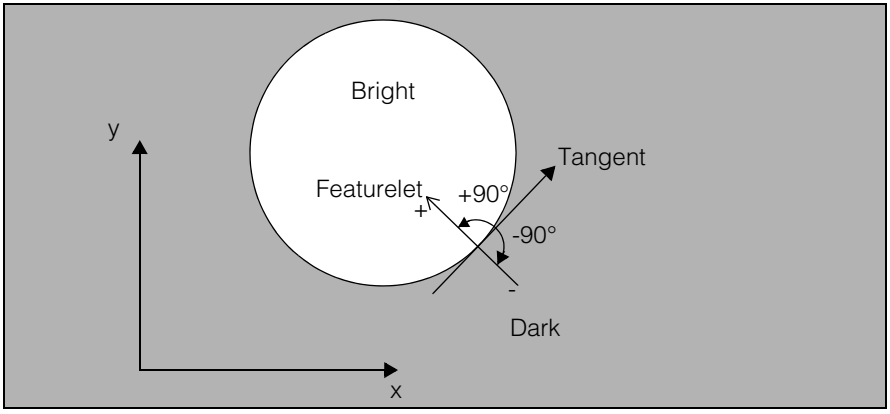
chainIndex The specified chain.

Throws

ccFeatureletDefs::BadIndex
If *chainIndex* < 0,
or if *chainIndex* >= **numChains()**.

Featurelet Chain Polarity

The CVL shapes convention defines the default polarity as *bright* in the direction of increasing angle with respect to the tangent direction along the shape boundary. The featurelet chain polarity follows this same convention assuming the featurelet angle corresponds to the gradient angle. See the following example of a circle where the featurelet positive direction is to the bright circle interior.



For closed featurelet chains, the inside/outside polarity corresponds to the combination of the **isRightHanded()** member function and the **proportionPositiveCrossProduct()** member function in the following way:

```
isNegativeOnPositive =
  ((isRightHanded() xor (proportionPositiveCrossProduct() >= 0.5)
```

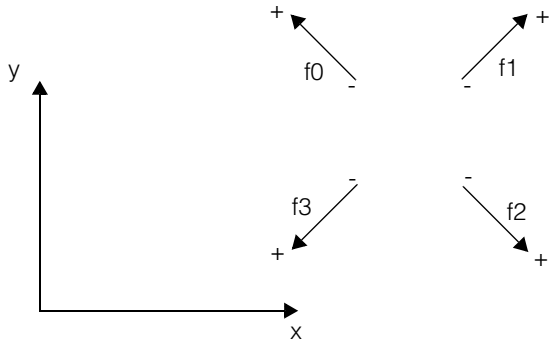
CVL provides a **ccPolylineModel** converter function which takes a featurelet chain set as its argument. (See **cfPolylineShapeModel()**). This converter computes reversed polarity using the **proportionPositiveCrossProduct()** member function in the following way:

```
isReversedPolarity == (proportionPositiveCrossProduct() < 0.5)
```

This implicitly assumes that the featurelet angles correspond to the gradient direction, from negative to positive.

The **proportionPositiveCrossProduction()** member function is used to determine polarity of a featurelet chain. This function assumes that the featurelet angles correspond to the gradient orientations from negative to positive.

Consider the closed featurelet chain (f0, f1, f2, f3) shown below where the arrows are pointing in the direction of the image gradient, from negative to positive.



For each featurelet, we compare the image gradient to both incident vectors.

Let a_i correspond to the unit vector in the direction of the angle at featurelet i, let f_i correspond to the position of featurelet i, and let isMod180_i correspond to whether featurelet i's orientation is defined modulo 180°. Let N be the number of featurelets. Then -

$$\textbf{proportionPositiveCrossProduction()} = \frac{1}{M} \times \sum_{i=0}^{N-1} \textit{if}\ (\text{isMod180\_\{i\}})\ \text{then 0,}$$

else sign((f_{i}-f{i-1}) cross a_i) + sign((f_{i+1}-f{i}) cross a_i)

Where $M = \sum_{i=0}^{N-1} \textit{if}\ (\text{isMod180\_\{i\}})\ \text{then 0, else 2,}$

and sign(x) = (+1 if x>0, 0 if x=0, or -1 if x<0).

Notes

The **proportionPositiveCrossProduction()** equation as written above is valid for closed chains only. For open chains, elements 1 and N each have only one incident vector.

Typedefs

ccFeatureletChainSetPtrh

```
typedef ccPtrHandle<ccFeatureletChainSet>
                                    ccFeatureletChainSetPtrh;
```

ccFeatureletChainSetPtrh_const

```
typedef ccPtrHandle_const<ccFeatureletChainSet>
                                  ccFeatureletChainSetPtrh_const;
```

ccFeatureletFilter

```
#include <ch_cvl/featfilt.h>

class ccFeatureletFilter: public virtual ccPersistent,
                          public virtual ccRepBase
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This is the abstract base class for featurelet filters and featurelet chain filters. Any filter class derived from this base class should behave as described in the section *Guidelines For Deriving New Filter Classes*, below.

Featurelet filtering with **ccFeatureletFilter** derived classes can be performed using the **filterFeaturelets()** member function and providing a vector of **ccFeaturelet** objects. Note that not all the filters provide featurelet filtering functionality and *ccFeatureletFilterDefs::FeatureletFilteringNotSupported* will be thrown if featurelet filtering is not supported.

Featurelet chain set filtering with **ccFeatureletFilter** derived classes can be performed using the **filterChains()** member function and providing a **ccFeatureletChainSet** object. Note that any filter derived from **ccFeatureletFilter** should provide this functionality.

Guidelines For Deriving New Filter Classes

You can design new filter classes easily by deriving from the **ccFeatureletFilter** class. To create your own filter class we suggest you copy and modify one of the standard filters provided in the *featfilt.h* header file.

The following rules are the guidelines for designing new filters.

The class constructor parameter for the *isInverted* flag (if there is one), should be passed to the base class function **isInverted()**.

The macro *cmFeatureletFilterClone* should be included in the public section of the class definition so that the **clone()** virtual member function is overridden.

The base class virtual function **filter_()** should be overridden with the desired functionality and it should assume the behavior listed under the **ccFeatureletFilter::filter_()** function.

The *survivorFlags* vector modified by the function **filter_()** should mark the filtered featurelet indices by false and its size should not be changed. For example, it should have the same size as the input vector *featurelets*.

The function **filter_()** should not perform any inversion operations based on the **isInverted()** flag. The filter inversions are done automatically by the base class member functions **filterChains()** and **filterFeaturelets()**.

Constructors/Destructors

ccFeatureletFilter

```
ccFeatureletFilter(bool isInverted = false);

virtual ~ccFeatureletFilter();
```

- `ccFeatureletFilter(bool isInverted = false);`

  Constructs a **ccFeatureletFilter** object with *isInverted* set as specified. Default is false.

  Parameters

isInverted	True or false.

- `virtual ~ccFeatureletFilter();`

  Destructor.

Operators

operator==

```
virtual bool operator== (
  const ccFeatureletFilter &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

operator!=

```
virtual bool operator!= (
  const ccFeatureletFilter &that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

clone

```
virtual ccFeatureletFilterPtrh clone() const = 0;
```

Returns a pointer to a new copy of this **ccFeatureletFilter**.

isInverted

```
void isInverted(bool t);

bool isInverted() const;
```

Use this function to set/get the inverted state of the featurelet filter.

- `void isInverted(bool t);`

  Sets the *isInverted* flag; true or false.

  Parameters

  t The new flag value.

- `bool isInverted() const;`

  Returns the current *isInverted* flag.

filterFeaturelets

```
void filterFeaturelets(
  const cmStd vector<ccFeaturelet>& inputFeaturelets,
  cmStd vector<ccFeaturelet>& outputFeatures) const;
```

Filters the input featurelets based on the criteria described by the derived class if **isInverted()** = false, and the inverse of the criteria if **isInverted()** = true.

Parameters

inputFeaturelets The featurelets to be filtered (input).

outputFeatures The filtered featurelets (output).

Throws

ccFeatureletFilterDefs::BadFilter
If the **filter_()** function is not properly overridden.

ccFeatureletFilterDefs::FeatureletFilteringNotSupported
If this filter does not support featurelet filtering.

Derived classes may have filter-specific throws.

Notes

The vector *outputFeatures* is resized to 0 before filtering starts.

See the protected **filter_()** function below for information on how to effectively override the behavior of this function in a derived class.

The behavior of this function is dependent on the derived filter class that calls it. Please see the documentation in the derived classes for filter-specific behavior of this function.

filterChains

```
ccFeatureletChainSetPtrh filterChains(
  const ccFeatureletChainSet& inputChainSet) const;
```

This function runs a filter on *inputChainSet* and returns the filtered result.

The function filters the featurelets in the given featurelet chain set based on the criteria described by the derived class if **isInverted()** is false, or the inverse of the criteria if **isInverted()** is true. The featurelets that remain after filtering are used to build a new chain set by eliminating, trimming or breaking the input chains. The resulting featurelet chains will be open unless the corresponding input chain is closed and none of its featurelets are eliminated.

Parameters

inputChainSet The chain set to filter.

Throws

ccFeatureletFilterDefs::BadFilter
If the **filter_()** function is not properly overridden.

Derived classes may have filter-specific throws.

Notes

See the protected **filter_()** function below for information on how to effectively override the behavior of this function in a derived class.

The behavior of this function is dependent on the derived filter class that calls it. Please see the documentation in the derived classes for filter-specific behavior of this function.

Protected Member Functions

filter_

```
virtual void filter_(
  const cmStd vector<ccFeaturelet> &featurelets,
  const cmStd vector<c_Int32> &featureletOrder,
```

```
const cmStd vector<c_Int32> &chainStartIndices,
const cmStd vector<bool> &closedFlags,
cmStd vector<bool> &survivorFlags) const = 0;
```

Notes

You should not call this function from your application code. This function is called internally by other filtering routines. If you create your own featurelet filter class by deriving from **ccFeatureletFilter** you will need to override this function in your new class. See examples of filters in the *featfilt.h* header file.

The following information is provided for those writing new filters.

Filters the featurelets in the chains for which the corresponding survivor flag is set, based on the criteria described by the derived class. Any featurelets that do not survive the filtering operation will have their corresponding survivor flags set to false. The order of featurelets is indicated by the featurelet order vector, which are indices into the featurelets. The start of each chain, i, is represented as an index, *chainStartIndices*[i], into the featureletOrder vector. For each chain, i, a closed flag is also specified by *closedFlags*[i]. Note that if this filter is performing a featurelet filtering operation (instead of featurelet chain set filtering), the vectors *featureletOrder*, *chainStartIndices*, and *closedFlags* are ignored because they provide information about the chains only. In order to be able to distinguish between two filtering types inside this function, this function assumes that the vectors *featureletOrder*, *chainStartIndices*, and *closedFlags* all have a size of 0 if it is performing a featurelet filtering operation. The following examples show how to obtain chain and featurelet information from the inputs to this function.

survivorFlags[j] is false if *featurelets*[j] is filtered out.

The following examples are valid only if this filter is performing featurelet chain set filtering.

featurelets[*featureletOrder*[*chainStartIndices*[i]+j]] accesses the jth element of the ith chain.

chainStartIndices[i+1] - *chainStartIndices*[i] returns the length of the ith chain, assuming that **chainStartIndices.size()** > i.

closedFlags[i] is true if ith chain is closed.

The following are requirements for this function:

The input vector element *survivorFlags*[j] should be true if and only if *featurelets*[j] was not filtered out previously by another filter.

The input vectors *survivorFlags* and *featurelets* should have the same size.

The vectors *featureletOrder*, *chainStartIndices*, and *closedFlags* all should have a size of 0 if this filter is performing a featurelet filtering operation.

The vectors *closedFlags* and *chainStartIndices* should have the same size.

The *chainStartIndices* all index within the bounds of the *featureletOrder* vector.

The *featureletOrder* all index within bounds of the *featurelets* vector.

featureletOrder does not contain two of any index.

Notes

Any derived class overriding this function can safely assume that all the requirements above are satisfied by the callers of this function.

Any derived class overriding this function should replicate the behavior above.

Throws

ccFeatureletFilterDefs::FeatureletFilteringNotSupported
If this filter does not support featurelet filtering and **featureletOrder.size()** is 0.

ccFeatureletFilterBoundary

```
#include <ch_cvl/featfilt.h>

class ccFeatureletFilterBoundary: public ccFeatureletFilter
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccFeatureletFilterBoundary is a filter class derived from **ccFeatureletFilter**. This filter provides functionality for filtering featurelets based on their positions, orientations, and polarities compared to a given **ccShape** boundary.

Using this filter class you can call **filterFeaturelets()** to filter featurelets, or **filterChains()** to filter featurelet chain sets. The behavior of these two functions when called from this class is described below.

Filter Featurelets

Featurelet filtering with **ccFeatureletFilterBoundary** can be performed using the **filterFeaturelets()** function by providing a vector of **ccFeaturelet** objects. First, **shape()** is sampled using the static member function **ccShapeModel::features()** with **featureParams()** in the class constructor. For each input featurelet, the sampled points within a distance less than, or equal to **distanceTol()** to the featurelet are found. If there are no such points, the featurelet is not filtered out (the featurelet survives). If there are such points, the angular differences between the tangents of these points and the featurelet tangent angle are measured. If any of these angular differences is less than, or equal to **angleTol()**, the featurelet is filtered out. Otherwise, the featurelet survives. The angular differences are taken in the range 0 through pi radians if **ignorePolarity()** is true. Otherwise, the angular differences are taken in the range 0 through 2pi to take the shape and featurelet polarity into consideration. If **angleTol()** is >= **ccRadian(**ckPI**)**, filtering based on angle tolerance will not be performed by turning off the *computeTangents* flag of *featureParams*. No filtering will be done for shapes if there are no tangent angles produced by sampling and **angleTol()** is not >= **ccRadian(**ckPI**)**.

Throws

ccFeatureletFilterDefs::BadParams
: If **ignorePolarity()** is false and **isMod180()** is true for any element of *inputFeaturelets*.

This function will not mask any throws from the static member function **ccShapeModel::features()**.

Filter Featurelet Chain Sets

Featurelet chain set filtering with **ccFeatureletFilterBoundary** can be performed using the **filterChains()** base class function by providing a **ccFeatureletChainSet** object. First, a featurelet filtering operation will be applied to the featurelets of the *inputChainSet*. The surviving featurelets from this filtering will be used to build a new **ccFeatureletChainSet**.

Throws

ccFeatureletFilterDefs::BadParams
If **ignorePolarity()** is false and **chainSet.anyFeatureletsIsMod180()** is true.

This function will not mask any throws from the static member function **ccShapeModel::features()**.

Constructors/Destructors

ccFeatureletFilterBoundary

```
explicit ccFeatureletFilterBoundary(
  const ccShape &shape = cc2Point(),
  double distanceTol = HUGE_VAL,
  const ccRadian &angleTol = ccRadian(ckPI),
  bool ignorePolarity = true,
  const ccShapeModel::ccFeatureParams &featureParams =
```

```
  ccShapeModel::ccFeatureParams (ccShape::ccSampleParams(
  0.0, 0.1, ckMaxInt32, true, false)),
bool isInverted = false);
```

Constructs a filter object with the given shape, distance and angle tolerance values, a flag to ignore featurelet polarity, and feature parameters. Default values are as follows:

Parameter	Default value
shape	**cc2Point()**
distanceTol	HUGE_VAL
angleTol	**ccRadian(**ckPI**)**
ignorePolarity	true
featureParams	sampling parameters: tolerance = 0.0 spacing = 0.1 max points = ckMaxInt32 compute tangents = true duplicate corners = false magScale = 1.0 weightScale = 1.0
isInverted	false

Please see **ccShapeModel** class for information on how to specify the shape polarity.

Parameters

shape The boundary geometric description.

distanceTol The maximum distance tolerance.

angleTol The maximum angle tolerance.

ignorePolarity If true, ignore feature chain polarity.

featureParams Shape model feature parameters.

isInverted If true filtering is inverted.

Throws

ccFeatureletFilterDefs::BadParams
If *distanceTol* < 0,
or if *angleTol* < 0,
or if *ignorePolarity* = false and the effective ignore polarity flag for any portion of the supplied shape is true.

This constructor will not mask any throws from the static member function **ccShapeModel::features()**.

Operators

operator==

```
virtual bool operator== (
  const ccFeatureletFilter &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

Public Member Functions

clone

```
Clone(ccFeatureletFilterBoundary);
```

Creates a new copy of this **ccFeatureletFilterBoundary** object.

Overrides the **clone()** function in the **ccFeatureletFilter** base class.

shape

```
ccShapePtrh_const shape() const;
```

Returns the shape specified during construction.

distanceTol

```
double distanceTol() const;
```

Returns the distance tolerance specified during construction.

angleTol

```
ccRadian angleTol() const;
```

Returns the angle tolerance specified during construction.

ignorePolarity

```
bool ignorePolarity() const;
```

Returns the *ignorePolarity* flag specified during construction.

featureParams

```
const ccShapeModel::ccFeatureParams &featureParams()
                                                    const;
```

Returns the *featureParams* specified during construction.

ccFeatureletFilterBoundary

ccFeatureletFilterComposite

```
#include <ch_cvl/featfilt.h>

class ccFeatureletFilterComposite: public ccFeatureletFilter
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccFeatureletFilterComposite is a filter class derived from **ccFeatureletFilter**. This filter provides functionality for filtering featurelets and featurelet chain sets by composing a vector of **ccFeatureletFilter**s you provide, into a single filter.

Using this filter class you can call **filterFeaturelets()** to filter featurelets, or **filterChains()** to filter featurelet chain sets. In each case, the function is called on each filter in the filter vector, in succession. The behavior of these two functions when called from this class is described below.

Filter Featurelets

Featurelet filtering with **ccFeatureletFilterComposite** can be performed using the **filterFeaturelets()** function and providing a vector of **ccFeaturelet** objects. The filtering performed is equivalent to, but more efficient than, calling the **filterFeatures()** function of each of the filters in order, each filter operating on the result of the previous filter.

Throws

ccFeatureletFilterDefs::FeatureletFilteringNotSupported
If any filter does not support featurelet filtering.

This function will not mask throws from filters you provide.

Filter Featurelet Chain Sets

Featurelet chain set filtering with **ccFeatureletFilterComposite** can be performed using the **filterChains()** function and providing a **ccFeatureletChainSet** object. The filtering performed is equivalent to, but more efficient than, calling the **filterChains()** function of each of the filters in order, each filter operating on the result of the previous filter.

Throws

This function will not mask throws from filters you provide.

Constructors/Destructors

ccFeatureletFilterComposite

```
explicit ccFeatureletFilterComposite(
  const cmStd vector<ccFeatureletFilterPtrh_const> &filters
         = cmStd vector<ccFeatureletFilterPtrh_const>(0),
  bool isInverted = false);
```

Constructs a filter object with the given filters and isInverted flag.

Parameters

filters	A vector of featurelet filters to run.
isInverted	If true, the filtering criteria is inverted.

Operators

operator==

```
virtual bool operator== (
  const ccFeatureletFilter &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

Public Member Functions

cmFeatureletFilterClone

```
cmFeatureletFilterClone(ccFeatureletFilterComposite);
```

A macro that creates a new copy of this **ccFeatureletFilterComposite** object.

Overrides the **clone()** function in the **ccFeatureletFilter** base class.

filters

```
const cmStd vector<ccFeatureletFilterPtrh_const>
                                          &filters() const;
```

Returns the filters specified during construction.

ccFeatureletFilterLength

```
#include <ch_cvl/featfilt.h>

class ccFeatureletFilterLength: public ccFeatureletFilter
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This filter class provides functionality for filtering featurelet chains based on their length, regardless of their open or closed status.

Using this filter class you can call **filterChains()** to filter featurelet chain sets. Calling **filterFeaturelets()** to filter featurelets is not supported by this filter. The behavior of these two functions when called from this class is described below.

Filter Featurelets

Featurelet filtering with this filter class is not allowed.

Throws

ccFeatureletFilterDefs::FeatureletFilteringNotSupported
If a featurelet filtering operation is attempted with this filter.

Filter Featurelet Chain Sets

You can perform featurelet chain set filtering with **ccFeatureletFilterLength** by calling the **filterChains()** function and providing a **ccFeatureletChainSet** object. Any chains with a length less than **minChainLength()** will not survive (they are filtered out) regardless of their open or closed status.

Constructors/Destructors

ccFeatureletFilterLength

```
explicit ccFeatureletFilterLength(
  c_Int32 minChainLength = 1,
  bool isInverted = false);
```

Constructor.

Parameters

minChainLength The minimum chain length required to survive the filter. Chains with length less than *minChainLength* are filtered out.

isInverted When set true, the filtering criteria is inverted. For example, all chain lengths less than *minChainLength* are retained, and others are filtered out.

Throws

ccFeatureletFilterDefs::BadParams
If *minChainLength* < 1.

Operators

operator==

```
virtual bool operator== (
  const ccFeatureletFilter &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

cmFeatureletFilterClone

```
cmFeatureletFilterClone(ccFeatureletFilterLength);
```

A macro that creates a new copy of this **ccFeatureletFilterLength** object.

Overrides the **clone()** function in the **ccFeatureletFilter** base class.

minChainLength

```
c_Int32 minChainLength() const;
```

Returns the minimum chain length specified during construction.

ccFeatureletFilterMagnitudeHysteresis

```
#include <ch_cvl/featfilt.h>

class ccFeatureletFilterMagnitudeHysteresis:
                                        public ccFeatureletFilter
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This filter class provides functionality for filtering featurelet chains based on hysteresis magnitude of the featurelets and their chains.

Using this filter class you can call **filterChains()** to filter featurelet chain sets. Calling **filterFeaturelets()** to filter featurelets is not supported by this filter. The behavior of these two functions when called from this class is described below.

Filter Featurelets

Featurelet filtering with this filter class is not allowed.

Throws

ccFeatureletFilterDefs::FeatureletFilteringNotSupported
If a featurelet filtering operation is attempted with this filter.

Filter Featurelet Chain Sets

You can perform featurelet chain set filtering with **ccFeatureletFilterMagnitudeHysteresis** by calling the **filterChains()** function and providing a **ccFeatureletChainSet** object. First, featurelets with a magnitude less than **lowMagThresh()** are filtered out and the surviving featurelets are used to build a temporary chain set. Second, the chains of this temporary set without at least one featurelet with a magnitude of greater than, or equal to **highMagThresh()** are filtered out. The remaining chains are returned as the filtered result.

Constructors/Destructors

ccFeatureletFilterMagnitudeHysteresis

```
explicit ccFeatureletFilterMagnitudeHysteresis(
  double lowMagThresh = 1,
  double highMagThresh = 1,
  bool isInverted = false);
```

Constructs an object with the given low and high magnitude threshold values.

Parameters

lowMagThresh	The featurelet low magnitude threshold.
highMagThresh	The featurelet high magnitude threshold.
isInverted	If true, filtering is inverted. Featurelets that survive when *isInverted*=false are filtered out, and the remaining featurelets survive.

Throws

ccFeatureletFilterDefs::BadParams
If *lowMagThresh* < 0.0 or *highMagThresh* < 0.0, or if *highMagThresh* < *lowMagThresh*.

Operators

operator==

```
virtual bool operator== (
  const ccFeatureletFilter &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

Public Member Functions

cmFeatureletFilterClone

```
cmFeatureletFilterClone(
  ccFeatureletFilterMagnitudeHysteresis);
```

A macro that creates a new copy of this **ccFeatureletFilterMagnitudeHysteresis** object.

Overrides the **clone()** function in the **ccFeatureletFilter** base class.

lowMagThresh `double lowMagThresh() const;`

Returns the low magnitude threshold specified during construction.

highMagThresh `double highMagThresh() const;`

Returns the high magnitude threshold specified during construction.

ccFeatureletFilterMagnitudeHysteresis

ccFeatureletFilterRegion

```
#include <ch_cvl/featfilt.h>

class ccFeatureletFilterRegion: public ccFeatureletFilter
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This filter provides functionality for filtering featurelets based on whether they are inside or outside of a geometric region.

Using this filter class you can call **filterFeaturelets()** to filter featurelets, or **filterChains()** to filter featurelet chain sets. The behavior of these two functions when called from this class is described below.

Filter Featurelets

Featurelet filtering with **ccFeatureletFilterRegion** can be performed using the **filterFeaturelets()** function and providing a vector of **ccFeaturelet** objects. Any featurelets outside the specified geometric region are filtered out.

Filter Featurelet Chain Sets

You can perform featurelet chain set filtering with **ccFeatureletFilterRegion** by calling the **filterChains()** function and providing a **ccFeatureletChainSet** object. A featurelet filtering operation will be applied to the featurelets of the *inputChainSet*. Any featurelets outside the specified geometric region are filtered out. The surviving featurelets from this filtering are then used to build a new **ccFeatureletChainSet**.

Constructors/Destructors

ccFeatureletFilterRegion

```
explicit ccFeatureletFilterRegion(
  const ccShape &region = ccCircle(),
  bool isInverted = false);
```

Constructs an object with the given region and *isInverted* flag.

Parameters

region The geometric region containing the featurelets of interest. Featurelets outside this region are filtered out (discarded).

isInverted If true, filtering is inverted. Featurelets outside the *region* are saved, and featurelets inside the *region* are filtered out.

Throws

ccShapesError::NotRegion
If **region.isRegion()** is false.

Operators

operator==

```
virtual bool operator== (
  const ccFeatureletFilter &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

cmFeatureletFilterClone

```
cmFeatureletFilterClone(ccFeatureletFilterRegion);
```

A macro that creates a new copy of this **ccFeatureletFilterRegion** object.

Overrides the **clone()** function in the **ccFeatureletFilter** base class.

region

```
ccShapePtrh_const region() const;
```

Returns the region specified during construction.

ccFeatureParams

```
#include <ch_cvl/shapemod.h>

class ccShapeModel::ccFeatureParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class specifies the parameters used to extract features from a shape model. An instance of this class is passed as a parameter to **ccShapeModel::features()** to describe how features should be extracted from the shape. This class specifies both shape model-specific parameters and shape sampling parameters. This is a nested class defined inside of **ccShapeModel**.

See the reference pages for **ccFeaturelet** and **ccFeatureletChainSet** for more information on featurelets and featurelet chain sets.

Constructors/Destructors

ccFeatureParams

```
ccFeatureParams(
  const ccShape::ccSampleParams& sampleParams =
    ccShape::ccSampleParams(0.0, 1.0, ckMaxInt32, true,
    true),
  double magScale = 1.0,
  double weightScale = 1.0);
```

Constructs a **ccShapeModel::ccFeatureParams** object with the given parameters.

Parameters

sampleParams	The shape sampling parameters.
magScale	The magnitude scale. Magnitude must be a non-negative value (default = 1.0).
weightScale	The weight scale. Weight can be any positive or negative value, or zero (default = 1.0).

Notes

The default parameters values cause the generated features to have the same magnitude and weight as those of the original shape. This may not be appropriate for all applications. For example, to generate features from a shape with a (default) magnitude of 1.0 and subsequently compare those features to featurelets generated using Sobel edge detection, *magScale* should be set to the edge contrast of the shape's boundary.

A default-constructed **ccFeatureParams** object specifies a feature density of at least one feature per unit of perimeter. This is often appropriate when the shape is defined in a coordinate system with the same scale as image coordinates. In other cases, non-default spacing parameters may be necessary.

Throws

ccShapeModelDefs::BadParams
magScale is less than 0.0.

Operators

operator== `bool operator==(const ccFeatureParams &rhs) const;`

Returns true if and only if this feature parameters object is exactly equal to *rhs*, otherwise false.

Parameters

rhs The other **ccFeatureParams** object.

operator!= `bool operator!=(const ccFeatureParams &rhs) const;`

Returns false if and only if this feature parameters object is exactly equal to *rhs*, otherwise true.

Parameters

rhs The other **ccFeatureParams** object.

Public Member Functions

sampleParams

```
const ccShape::ccSampleParams& sampleParams() const;
void sampleParams(const ccShape::ccSampleParams& params);
```

- `const ccShape::ccSampleParams& sampleParams() const;`

  Retrieves the shape sampling parameters. The values for a default-constructed **ccShapeModel::ccFeatureParams** object are as follows:

• *tolerance*	0.0 (no tolerance)
• *spacing*	1.0
• *maxPoints*	*ckMaxInt32* (no maximum)
• *computeTangents*	true
• *duplicateCorners*	true

- `void sampleParams(const ccShape::ccSampleParams& params);`

  Sets the shape sampling parameters.

  Parameters

  params The shape sampling parameters.

magScale

```
double magScale() const;
void magScale(double val);
```

- `double magScale() const;`

  Retrieves the magnitude scale. This value is multiplied by the effective magnitude of each portion of the sampled shape model to determine the magnitudes of the respective featurelets sampled from those portions. The default magnitude scale is 1.0.

- `void magScale(double val);`

  Sets the magnitude scale. This value is multiplied by the effective magnitude of each portion of the sampled shape model to determine the magnitude of the respective featurelets sampled from those portions.

Throws

ccShapeModelDefs::BadParams
val is less than 0.0.

Notes

In order to generate features from a shape with a (default) magnitude of 1.0 and subsequently compare those features to featurelets generated using Sobel edge detection, *magScale* should be set to the edge contrast of the shape's boundary.

---

weightScale

```
double weightScale() const;
void weightScale(double val);
```

- `double weightScale() const;`

  Retrieves the weight scale. This value is multiplied by the effective weight of each portion of the sampled shape model to determine the weights of the respective featurelets sampled from those portions. The default weight scale is 1.0.

- `void weightScale(double val);`

  Sets the weight scale. This value is multiplied by the effective weight of each portion of the sampled shape model to determine the weights of the respective featurelets sampled from those portions.

ccFeatureSegment

```
#include <ch_cvl/pmifproc.h>

class ccFeatureSegment : public cc_FeatureRange;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that describes a single feature segment. A feature segment is a contiguous range of a feature where the difference in direction between neighboring feature boundary points is small.

You do not instantiate **ccFeatureSegment**s yourself. You can segment a feature into a vector of **ccFeatureSegment**s by calling **cfSegmentFeature()**.

Constructors/Destructors

ccFeatureSegment

```
ccFeatureSegment();
```

Constructs a **ccFeatureSegment**.

Public Member Functions

length

```
double length() const;
```

Return the length of this feature segment in client coordinate system units. The length of a feature is the sum of the distances between the feature's boundary points.

weight

```
double weight() const;
```

Returns the average weight of all of the feature boundary points that make up this feature segment.

matchQuality

```
double matchQuality() const;
```

Returns the average match quality of all of the feature boundary points that make up this feature segment.

ccFeatureSegment

ccFileArchive

```
#include <ch_cvl/archive.h>

class ccFileArchive : public ccArchive;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	Not applicable

The **ccFileArchive** class is a concrete class derived from **ccArchive** that implements a file archive.

Constructors/Destructors

ccFileArchive

```
ccFileArchive(const ccCvlString& fileName,
  ccArchive::Direction dir,
  ccArchive::Ordering ord = ccArchive::eLittleEndian,
  bool seekable=true, bool readOnly=false);

~ccFileArchive();
```

- ```
  ccFileArchive(const ccCvlString& fileName,
    ccArchive::Direction dir,
    ccArchive::Ordering ord = ccArchive::eLittleEndian,
    bool seekable=true, bool readOnly=false);
  ```

  Constructs a file archive associated with the specified file.

  If the archive is constructed for storing, the *readOnly* argument is ignored. If the file already exists, it is truncated.

  Parameters

fileName	The file that contains the archive.
dir	The mode of this archive (loading or storing). *dir* must be one of the following values: *ccArchive::eLoading* *ccArchive::eStoring*
  ```

ord — The endianness of this archive. *ord* must be one of the following values:

ccArchive::eBigEndian
ccArchive::eLittleEndian

seekable — If true, the seek position within this archive can be specified.

readOnly — If true, this archive cannot be stored to.

You must set *readOnly* to true if you have read-only access to the file containing the archive.

Throws

ccArchive::BadFile
The specified file could not be opened. Verify that the file exists.

ccArchive::UnknownArchiveVersion
The loading archive was stored with a later version.

Notes

Only one **ccFileArchive** may be associated with a given file at a time.

ccFilterConvolveParams

```
#include <ch_cvl/filterconv.h>

class ccFilterConvolveParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the parameters used by the variable-kernel discrete convolution tool.

Constructors/Destructors

ccFilterConvolveParams

```
ccFilterConvolveParams();
```

Public Member Functions

boundaryMode

```
ccFilterDefs::BoundaryMode boundaryMode() const ;

void boundaryMode(ccFilterDefs::BoundaryMode rhs);
```

- `ccFilterDefs::BoundaryMode boundaryMode() const;`

  Returns the current boundary mode. The returned value is one of the following:

  ccFilterDefs::eBoundaryClipped
  ccFilterDefs::eBoundaryWeighted
  ccFilterDefs::eBoundaryReflected

- `void boundaryMode(ccFilterDefs::BoundaryMode rhs);`

  Sets the boundary mode for convolution. The default mode is *ccFilterDefs::eBoundaryClipped*, where ouput pixels are only computed for locations where the kernel is fully contained within the bounds of the source image, resulting in an output image that is smaller than the input image by the size of the kernel minus 1.

Parameters

rhs The boundary mode to set. Must be *ccFilterDefs::eBoundaryClipped* or *ccFilterDefs::eBoundaryReflected*

Throws

ccFilterDefs::NotImplemented
A mode other than *ccFilterDefs::eBoundaryClipped* or *ccFilterDefs::eBoundaryReflected* was specified.

operator==

```
bool operator==(const ccFilterConvolveParams& rhs) const;
```

Return true if two objects are equal.

Parameters

rhs The object to compare to this one.

operator!=

```
bool operator!=(const ccFilterConvolveParams& rhs) const;
```

Return true if two objects are not equal.

Parameters

rhs The object to compare to this one.

ccFilterConvolveKernel

```
#include <ch_cvl/filterconv.h>

template <class P>class ccFilterConvolveKernel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the kernel used by the variable-kernel discrete convolution tool. This object is deep copied upon copy construction or assignment.

Constructors/Destructors

ccFilterConvolveKernel

```
ccFilterConvolveKernel();
```

Construct an object with uninitialized data.

Notes

cfFilterConvolve will throw *ccFilterDefs::UnboundKernel* if uninitialized kernel is used.

Public Member Functions

kernel

```
const ccPelBuffer_const<K>& kernel() const;

void kernel(const ccPelBuffer_const<K>& rhs);
```

- ```
  const ccPelBuffer_const<K>& kernel() const;
  ```

  Returns the kernel image.

- ```
  void kernel(const ccPelBuffer_const<K>& rhs);
  ```

  Sets the kernel image. A deep copy of the supplied image is made.

  Parameters

rhs	The kernel image.

Throws

ccFilterDefs::UnboundKernel

The specified kernel is unbound.

ccFilterDefs::BadKernelSize

Either the width or height of the specified kernel is not a positive odd integer.

ccFilterDefs

```
#include <ch_cvl/filtercomm.h>

class ccFilterDefs;
```

Enumerations

BoundaryMode How the tool applies the kernel when the neighborhood is not fully contained within the source image.

Value	Meaning
eBoundaryClipped	No output pixel value is calculated if the kernel extends outside the source image. Consequently, the output image is reduced in width and height relative to the input image by the kernel width minus one and the kernel height minus 1.
eBoundaryWeighted	Only the pixel values that lie inside the source image are used to compute the output pixel value. The result of the computation is weighted to reflect the number of pixels used to compute the value relative to the total kernel size.
eBoundaryReflected	The edge of the input image is padded with pixel values generated by reflecting the values present at the boundary of the input image.

ccFilterDefs

ccFilterMaskKernel

```
#include <ch_cvl/filtercomm.h>

class ccFilterMaskKernel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class describes the kernel used by the variable-kernel median filter tool.

Constructors/Destructors

ccFilterMaskKernel

```
explicit ccFilterMaskKernel(
     const ccIPair& size = ccIPair(3, 3));

explicit ccFilterMaskKernel(
   const ccPelBuffer_const<c_UInt8>& mask);
```

- ```
  explicit ccFilterMaskKernel(
       const ccIPair& size = ccIPair(3, 3));
  ```

  Constructs a kernel of the specified size (width, height). The kernel mask is set filled with care pixels.

  Parameters

  size A **ccIPair** containing the width (*size.X*) and height (*size.Y*) of the kernel. Both the width and height must be positive odd integers.

  Notes

  The x-component denotes the width of the kernel, and y-component the height.

  Throws

  ccFilterDefs::BadKernelSize

  If either the width or height of the given size is not a positive odd integer.
  ```

- ```
  explicit ccFilterMaskKernel(
      const ccPelBuffer_const<c_UInt8>& mask);
  ```

  Construct the kernel using the supplied image. The width and height of the kernel are set to the width and height of *mask*, and the mask values are set to the pixel values of the supplied image.

  Parameters

  mask An image defining the kernel size and containing the mask values.

  Throws

  ccFilterDefs::BadKernelSize
  The width or height of *mask* is not a positive odd integer.

Public Member Functions

size

```
const ccIPair size() const;

void size(const ccIPair& rhs);
```

- ```
  const ccIPair size() const;
  ```

  Returns a **ccIPair** giving the size of this kernel. The x-component gives the width, the y-component the height.

- ```
  void size(const ccIPair& rhs);
  ```

  Sets the size of this kernel to that given by the supplied **ccIPair**. The x-component gives the width, the y-component the height. Both the width and height must be positive odd integers.

  The mask image is resized to the new kernel size. If the new size is larger in either width or height than the current size, the newly added mask pixels are set to care (255).

  Parameters

  rhs The size to set.

  Throws

  ccFilterDefs::BadKernelSize
  The width or height of *rhs* is not a positive odd integer.
  ```

mask

```
const ccPelBuffer_const<c_UInt8>& mask() const;
void mask(const ccPelBuffer_const<c_UInt8>& rhs);
```

- `const ccPelBuffer_const<c_UInt8>& mask() const;`

  Returns the kernel's mask image.

- `void mask(const ccPelBuffer_const<c_UInt8>& rhs);`

  Sets both the kernel and the mask image based on the supplied image. The kernel is reset to the height and width of the supplied image and the mask image is set to the pixel values of the supplied image.

  Parameters

  rhs The image from which to set this kernel

  Throws

  ccFilterDefs::BadKernelSize
  The width or height of the supplied image is not a positive odd integer.

  Notes

  The offset and client coordinates of the mask image are ignored. The origin of the kernel is always at the center of the kernel.

  The supplied image is deep copied.

Operators

operator=

```
const ccFilterMaskKernel& operator=(
    const ccFilterMaskKernel& rhs);
```

Assignment operator

Parameters

rhs The **ccFilterMaskKernel** to assign.

Notes

The mask image is deep copied.

operator==

```
bool operator==(const ccFilterMaskKernel& rhs) const;
```

Return true if the two kernels, including mask images, are equal. The mask images are compared by **cfPelEqual()**.

Parameters

rhs The kernel to compare to this one.

operator!=

```
bool operator!=(const ccFilterMaskKernel& rhs) const;
```

Return true if the two kernels, including mask images, are not equal. The mask images are compared by **cfPelEqual()**.

Parameters

rhs The kernel to compare to this one.

ccFilterMedianParams

```
#include <ch_cvl/filtmed.h>

class ccFilterMedianParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the parameters used by the variable-kernel median filter tool.

Constructors/Destructors

ccFilterMedianParams

```
ccFilterMedianParams();
```

Compiler-generated copy constructor and assignment operator

Public Member Functions

boundaryMode

```
ccFilterDefs::BoundaryMode boundaryMode() const;

void boundaryMode(ccFilterDefs::BoundaryMode rhs);
```

- `ccFilterDefs::BoundaryMode boundaryMode() const;`

  Returns the boundary mode used by this tool. This function always returns *ccFilterDefs::eBoundaryWeighted*.

- `void boundaryMode(ccFilterDefs::BoundaryMode rhs);`

  Sets the boundary mode.

  Parameters

  rhs The mode to set. Must be *ccFilterDefs::eBoundaryWeighted.*

  Throws

  ccFilterDefs::NotImplemented
  rhs is not *ccFilterDefs::eBoundaryWeighted*.

Operators

operator== `bool operator==(const ccFilterMedianParams& rhs) const;`

Return true if two objects are equal

Parameters

rhs The object to compare to this one.

operator!= `bool operator!=(const ccFilterMedianParams& rhs) const;`

Return true if two objects are not equal

Parameters

rhs The object to compare to this one.

ccFilterMorphologyStructuringElement

```
#include <ch_cvl/filtmrph.h>

class ccFilterMorphologyStructuringElement;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the structuring element used by the variable-kernel grey-scale morphology tool. Both the element and the mask image are deep copied upon copy construction or assignment.

Constructors/Destructors

ccFilterMorphologyStructuringElement

```
ccFilterMorphologyStructuringElement();

ccFilterMorphologyStructuringElement(
     const ccGMorphElement& gmorphElement);

ccFilterMorphologyStructuringElement
     (const ccFilterMorphologyStructuringElement& rhs);
```

- `ccFilterMorphologyStructuringElement();`

  Construct an object with uninitialized data.

  Notes

  This tool will throw *ccFilterDefs::UnboundKernel* if an uninitialized structuring element is used

- ```
  ccFilterMorphologyStructuringElement(
       const ccGMorphElement& gmorphElement);
  ```

  Construct an equivalent structuring element to the given **ccGMorphElement** object (used by the fixed-kernel tool).
  ```

Parameters

gmorphElement The element from which to construct this **ccFilterMorphologyStructuringElement**.

Notes

If the resulting structuring element uses a kernel offset, this constructor sets *useOffsetImage* to *ceTrue*.

- `ccFilterMorphologyStructuringElement (const ccFilterMorphologyStructuringElement& rhs);`

Copy constructor.

Parameters

rhs The source of the copy.

Notes

The constructor makes a deep copy of the *rhs*'s mask and *offsetImage*.

---

mask

```
const ccPelBuffer_const<c_UInt8>& mask() const;

void mask(const ccPelBuffer_const<c_UInt8>& rhs);
```

- `const ccPelBuffer_const<c_UInt8>& mask() const;`

Returns the kernel mask image.

- `void mask(const ccPelBuffer_const<c_UInt8>& rhs);`

Sets the kernel mask image. only Structuring element pixels corresponding to care (255) mask pixels are used to compute the output image; don't care pixels (0) are ignored.

You can specify any pattern of care and don't care pixels so as as to acheive a non-rectangular element.

The supplied image is deep copied.

Parameters

rhs The mask image.

Throws

ccFilterDefs::BadKernelSize
The width or height of rhs is not a positive odd integer.

Notes

The offset and client coordinates of the mask image are ignored. The origin of the kernel is always at the center of the kernel.

The function makes a deep copy of the given image.

useOffsetImage

```
c_Bool useOffsetImage() const;
void useOffsetImage(c_Bool use);
```

- `c_Bool useOffsetImage() const;`

  Gets whether or not to use the *offsetImage*, if one is specified.

- `void useOffsetImage(c_Bool use);`

  Sets whether or not to use the *offsetImage*, if one is specified.

  The default value is *ceFalse*.

offsetImage

```
const ccPelBuffer_const<c_Int16>& offsetImage();
void offsetImage(const ccPelBuffer_const<c_Int16>& rhs);
```

- `const ccPelBuffer_const<c_Int16>& offsetImage();`

  Returns a 16-bit image containing the structuring element values.

- `void offsetImage(const ccPelBuffer_const<c_Int16>& rhs);`

  Sets the structuring element values to those in the supplied image. The pixels are deep copied.

  Parameters

  rhs The image to use.

  Throws

  ccFilterDefs::BadMask
  A pixel values in *rhs* is outside the [-255, 255] range.

Notes

If *useOffsetImage* is false, then **cfFilterMorphology()** ignores this value.

The function makes a deep copy of the given image.

The offset and client coordinates of the given image are ignored.

size

```
const ccIPair size() const;
void size(const ccIPair& size);
```

- `const ccIPair size() const;`

Returns the size of the structuring element. **size().x** is the width while **size().y** is the height.

- `void size(const ccIPair& size);`

Sets the size of the structuring element to the supplied values. size.x gives the wdith while size.y gives the height.

Parameters

size The size to set.

Throws

ccFilterDefs::BadKernelSize
Either the width or height of the given size is not a positive odd integer.

Notes

The mask is reset to the supplied size and filled with care pixels (255) while the offset image is set to unbound.

The default is (0,0).

Operators

operator==

```
bool operator==(
   const ccFilterMorphologyStructuringElement& rhs) const;
```

Return true if two objects are equal. The mask and offset images are compared using **cfPelEqual()**.

Parameters

rhs The object to compare with this one.

operator!=

```
bool operator!=(const
ccFilterMorphologyStructuringElement& rhs) const;
```

Return true if two objects are not equal. The mask and offset images are compared using **cfPelEqual()**.

Parameters

rhs The object to compare with this one.

operator=

```
const ccFilterMorphologyStructuringElement& operator
=(const ccFilterMorphologyStructuringElement& rhs);
```

Assignment operator.

Parameters

rhs The source of the assignment.

Notes

The operator makes a deep copy of the *rhs*'s mask and *offsetImage*.

ccFilterMorphologyStructuringElement

ccFilterMorphologyDefs

```
#include <ch_cvl/filtmorph.h>
```

Enumerations

Operation

The morphological operation to perform.

Value	Meaning
eDilate	Expand light areas and shrink dark areas in an image..
eErode	Shrink light areas and expand dark areas in an image.
eOpen	Erode and then dilate with the same structuring element. Eliminate extraneous light details, thin lines, and small islands. Smooth object contours while maintaining dark holes or narrow channels.
eClose	Dilate and then erode with the same structuring element. Eliminate small dark holes and gaps in light objects, and smooth edges while preserving image details.

ccFilterMorphologyDefs

ccFilterMorphologyParams

```
#include <ch_cvl/filtmorph.h>

class ccFilterMorphologyParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the parameters used by the variable-kernel grey-scale morphology tool.

Constructors/Destructors

ccFilterMorphologyParams

```
ccFilterMorphologyParams();
```

Compiler-generated copy constructor and assignment operator.

Public Member Functions

boundaryMode

```
ccFilterDefs::BoundaryMode boundaryMode() const;

void boundaryMode(ccFilterDefs::BoundaryMode rhs);
```

- ```
  ccFilterDefs::BoundaryMode boundaryMode() const;
  ```

  Returns the boundary mode used by this tool. Always returns *ccFilterDefs::eBoundaryWeighted*.

- ```
  void boundaryMode(ccFilterDefs::BoundaryMode rhs);
  ```

  Sets the boundary mode.

  Parameters

rhs	The boundary mode. Must be *ccFilterDefs::eBoundaryWeighted*.

Throws

ccFilterDefs::NotImplemented

rhs is not *ccFilterDefs::eBoundaryWeighted.*

Operators

operator==

```
bool operator==(const ccFilterMorphologyParams& rhs) const;
```

Return true if two instances are equal

Parameters

rhs	The object to compare to this one.

operator!=

```
bool operator!=(const ccFilterMorphologyParams& rhs) const;
```

Return true if two instances are not equal

Parameters

rhs	The object to compare to this one.

ccFirstPelOffsetProp

```
#include <ch_cvl/prop.h>

class ccFirstPelOffsetProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

Analog line scan cameras do not provide the acquisition software with pixel clock information, which makes it difficult to determine exactly when an acquisition should begin. In such cases, acquired images may contain black pixels at the beginning or end of an image. This is not an issue with digital line scan cameras.

This property allows you to specify the offset of the first image pixel from the value supplied by the camera's device driver.

Do not use this class to adjust the region of interest for an acquisition. To specify a region of interest use the class **ccRoiProp** on page 2763.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccFirstPelOffsetProp

```
ccFirstPelOffsetProp();

explicit ccFirstPelOffsetProp(c_Int8 offset);
```

- `ccFirstPelOffsetProp();`

  Creates a new first pixel offset property not associated with any FIFO. The default offset is used.

- `explicit ccFirstPelOffsetProp(c_Int8 offset);`

  Creates a new first pixel offset property not associated with any FIFO. The offset is specified by *offset*.

  Parameters

  offset The first pixel offset in pixels.

Public Member Functions

firstPelOffset

```
void firstPelOffset(c_Int8 offset);
c_Int8 firstPelOffset() const;
```

- `void firstPelOffset(c_Int8 offset);`

  Sets the first pixel offset.

  Parameters

  offset The first pixel offset in pixels.

  Throws

  ccFirstPelOffsetProp::BadParams
  offset was not in the range zero through *ccFirstpelOffsetProp::maxDelayOffset*.

- `c_Int8 firstPelOffset() const;`

  Gets the first pixel offset in pixels.

Constants

defaultDelayOffset

```
static const c_Int8 defaultDelayOffset;
```

The default delay offset: zero.

maxDelayOffset

```
static const c_Int8 maxDelayOffset;
```

The default maximum delay offset: 127.

ccFLine

```
#include <ch_cvl/shapes.h>

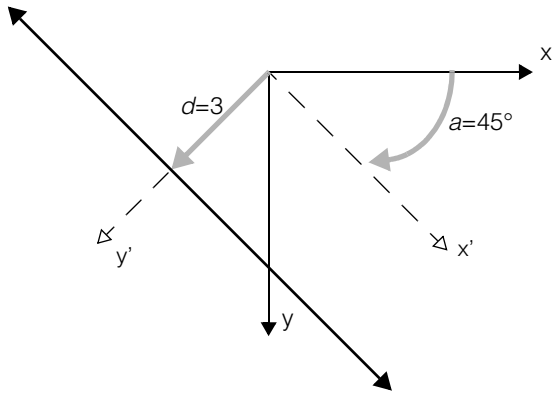
class ccFLine : public ccShape;
```

Class Properties

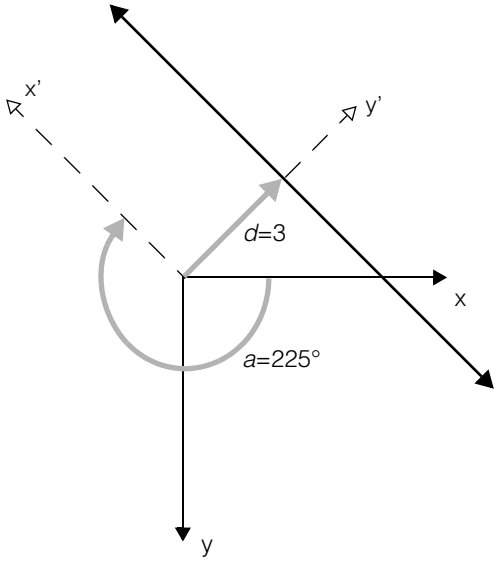
Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a line rotated by a given angle *a* relative to the x-axis and translated a specified distance *d* from the origin. *a* can be positive or negative. *d* is always positive. The line is interpreted as if it were parallel to the x-axis of a coordinate frame that is rotated *a* degrees, and translated along the rotated y-axis by *d*.

The following figure shows a line that is rotated 45° and has a distance of 3.



The following figure shows a line rotated 225° with a distance of 3.



Notes

Once created, a **ccFLine** can be modified only by the assignment operator.

Constructors/Destructors

ccFLine

```
ccFLine();
ccFLine(const ccDegree& angle, double distance);
ccFLine(const cc2Vect & point1, const cc2Vect & point2);
ccFLine(const ccDegree & angle, const cc2Vect & point);
```

- `ccFLine();`

  The default constructor creates a line whose angle and distance are both zero. The line is colinear with the x-axis.

- `ccFLine(const ccDegree& angle, double distance);`

  Creates a line rotated *angle* degrees from the x-axis and *distance* from the origin.

Parameters

angle The angle of the line.

distance The distance between the origin and the line. Must be nonnegative.

Throws

ccShapesError::DegenerateShape
distance is less than zero.

- `ccFLine(const cc2Vect & point1, const cc2Vect & point2);`

Creates the line that includes two points.

Parameters

point1 A first point in the line.

point2 A second point in the line.

Throws

ccShapesError::DegenerateShape
point1 equals *point2*.

- `ccFLine(const ccDegree & angle, const cc2Vect & point);`

Creates the line rotated *angle* degrees from the x-axis that includes *point*.

Parameters

angle The angle of the line.

point A point on the line.

Operators

operator== `int operator== (const ccFLine& line) const;`

Returns true if this line has the same angle and distance as another line.

Parameters

line The other line.

operator!= `int operator!= (const ccFLine& line) const;`

Returns true if this line is not equal to another line.

Parameters

line	The other line.

operator*

```
friend ccFLine operator* (const cc2Xform& xform,
 const ccFLine & line);

friend ccFLine operator* (const cc2Rigid& xform,
 const ccFLine & line);
```

- ```
  friend ccFLine operator* (const cc2Xform& xform,
   const ccFLine & line);
  ```

  Maps *line* by *xform* and returns a new line.

  Parameters

xform	The **cc2Xform** being applied.
line	An existing line.

- ```
  friend ccFLine operator* (const cc2Rigid& xform,
   const ccFLine & line);
  ```

  Maps *line* by *xform* and returns a new line.

  Parameters

xform	The **cc2Rigid** being applied.
line	An existing line.

operator<<

```
friend ccCvlOStream& operator<< (ccCvlOStream& out,
 const ccFLine& line);
```

Prints the supplied **ccFLine**.

Parameters

out	The stream to which to print the line.
line	The **ccFLine** to print.

Public Member Functions

angle

```
const ccDegree& angle();
```

Returns the line's angle.

distance `double distance() const;`

Returns the line's distance from the origin.

sinAngle `double sinAngle() const;`

Returns the sine of the line's angle.

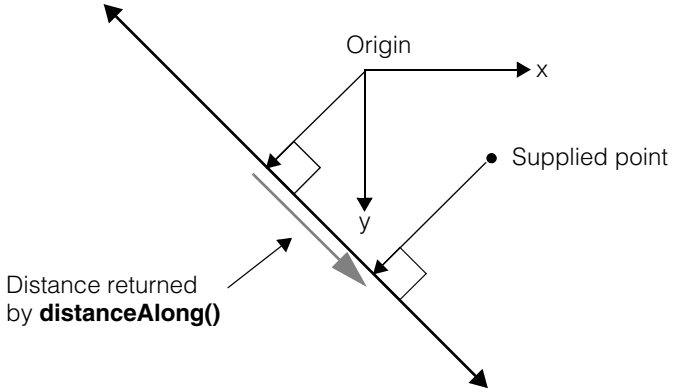
cosAngle `double cosAngle() const;`

Returns the cosine of the line's angle.

distanceAlong `double distanceAlong(const cc2Vect & point) const;`

Returns the signed distance from the projection of the origin point on the line to the projection of the supplied point on the line. The positive direction is specified by the line's angle.

The following figure shows the value returned by this function:



Parameters

point An arbitrary point.

intersection `cmStd vector<cc2Vect> intersection(const ccFLine & line) const;`

Returns the single point of intersection between this line and another line if an intersection exists, or an empty vector if no intersection exists.

Parameters

other Another line.

map `ccFLine map(const cc2Xform& c) const;`

Returns this line mapped by *c*.

Parameters

c The transformation object.

clone `virtual ccShapePtrh clone() const;`

Returns a pointer to a copy of this line.

isOpenContour `virtual bool isOpenContour() const;`

Returns true if this shape is an open contour. For lines, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion `virtual bool isRegion() const;`

Returns true if this shape is a region. For lines, this function always returns false. See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

For lines, this function always returns false. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For lines, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

For lines, this function always returns true. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For lines, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For lines, this function always returns false.

Notes

ccFLines are not reversible, but **ccLine**s are.

See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Throws

ccShapesError::InfiniteExtent
For all **ccFLine**s.

See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on this line to the given point.

Parameters

p The point.

See **ccShape::nearestPoint()** for more information.

perimeter

```
virtual double perimeter() const;
```

Throws

ccShapesError::InfiniteExtent
For all **ccFLine**s.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this line to the given point, as determined by **nearestPoint()**.

Parameters

info Shape information for this line.

point The point.

See **ccShape::nearestPerimPos()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Throws

ccShapesError::InfiniteExtent
For all **ccFLine**s.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this line mapped by *X*.

Parameters

X	The transformation object.

Notes

If *X* is singular, the mapping may collapse this line to a single point.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccGeneralShapeTree** without any children. See **ccShape::decompose()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the line segment describing the portion of this line over the given range. The length of the final returned line segment is equal to the absolute value of the distance component of *range*.

Parameters

info	Shape information for this line.
range	The perimeter range.

See **ccShape::subShape()** for more information.

Static Functions

Xaxis

```
static ccFLine Xaxis(const cc2Rigid & xform);
static ccFLine Xaxis(const cc2Xform & xform);
```

- `static ccFLine Xaxis(const cc2Rigid & xform);`

  Returns the x-axis of a **cc2Rigid** transform as a line.

  Parameters
  xform A **cc2Rigid** transform.

- `static ccFLine Xaxis(const cc2Xform & xform);`

  Returns the x-axis of a **cc2Xform** transform as a line.

  Parameters
  xform A **cc2Xform** transform.

Yaxis

```
static ccFLine Yaxis(const cc2Rigid & xform);
static ccFLine Yaxis(const cc2Xform & xform);
```

- `static ccFLine Yaxis(const cc2Rigid & xform);`

  Returns the y-axis of a **cc2Rigid** transform as a line.

  Parameters
  xform A **cc2Rigid** transform.

- `static ccFLine Yaxis(const cc2Xform & xform);`

  Returns the y-axis of a **cc2Xform** transform as a line.

  Parameters
  xform A **cc2Xform** transform.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

ccFLine

closestPoint `cc2Vect closestPoint(const cc2Vect & point) const;`

Use **nearestPoint()** instead of this function.

distanceFrom `double distanceFrom(const cc2Vect & point) const;`

Use **distanceToPoint()** instead of this function.

ccFontCharMetrics

```
#include <ch_cvl/synfont.h>

class ccFontCharMetrics;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccFontCharMetrics** class contains various metrics for a single character.

Constructors/Destructors

ccFontCharMetrics

```
ccFontCharMetrics();
```

Constructs a default character metrics object.

Notes

The default copy constructor, assign operator, and destructor are used.

Public Member Functions

cellRect

```
ccRect cellRect() const;

void cellRect(const ccRect& r);
```

- ```
  ccRect cellRect() const;
  ```

  Gets cell rect.

- ```
  void cellRect(const ccRect& r);
  ```

  Sets cell rect.

  The default is **ccRect()**

  Parameters

  r The new cell rect.

isMarkRectSpecified

```
bool isMarkRectSpecified() const;
void isMarkRectSpecified(bool s);
```

- `bool isMarkRectSpecified() const;`

  Gets whether the mark rect is specified.

- `void isMarkRectSpecified(bool s);`

  Sets whether the mark rect is specified.

  The default is false.

  Parameters

  s If true, the mark rect is specified.

  Notes

  A blank character has a mark rect that is specified but degenerate.

markRect

```
ccRect markRect() const;
void markRect(const ccRect& r);
```

- `ccRect markRect() const;`

  Gets mark rect.

  Throws

  ccSynFontDefs::MarkRectNotSpecified
  isMarkRectSpecified is false.

- `void markRect(const ccRect& r);`

  Sets mark rect.

  The default is **ccRect()**

  Parameters

  r The new mark rect.

hasDetectedMarkRect

```
bool hasDetectedMarkRect() const;
void hasDetectedMarkRect(bool s);
```

- `bool hasDetectedMarkRect() const;`

  Gets whether there is a detected mark rect.

- `void hasDetectedMarkRect(bool s);`

  Sets whether there is a detected mark rect.

  The default is false.

  Parameters

  s If true, there is a detected mark rect.

  Notes

  A blank character does not have a detected mark rect.

detectedMarkRect

```
ccRect detectedMarkRect() const;
void detectedMarkRect(const ccRect& r);
```

- `ccRect detectedMarkRect() const;`

  Gets the detected mark rect.

  Throws

  ccSynFontDefs::MarkRectNotSpecified
  The **hasDetectedMarkRect()** operation returned false.

- `void detectedMarkRect(const ccRect& r);`

  Sets the detected mark rect.

  The default is **ccRect()**

  Parameters

  r The detected mark rect.

isBlank

```
bool isBlank() const;
void isBlank(bool b);
```

- `bool isBlank() const;`

  Gets whether or not the character is a blank (space).

- `void isBlank(bool b);`

  Sets whether or not the character is a blank (space).

  Setting *isBlank* to true is the same as specifying a degenerate mark rect; getter is the same as testing for a mark rect that is specified and degenerate.

  Parameters

  b — If true, the character is a blank (space).

advance

```
cc2Vect advance() const;
void advance(const cc2Vect& a);
```

- `cc2Vect advance() const;`

  Gets the advance for the character.

- `void advance(const cc2Vect& a);`

  Sets the advance for the character.

  The default is **cc2Vect(**0, 0**)**

  Parameters

  a — The new advance for the character.

ccFontKey

```
#include <ch_cvl/synfont.h>

class ccFontKey;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **ccFontKey** class represents a character key, which consists of a 16-bit Unicode code point and a 16-bit instance number. The instance number allows supporting fonts that have multiple representations of a given character.

Note **ccFontKey** does not support 21-bit Unicode code points.

Constructors/Destructors

ccFontKey

```
ccFontKey(c_Int32 key = 0): key_(key);

ccFontKey(TCHAR charCode, c_Int32 instance);
```

- `ccFontKey(c_Int32 key = 0): key_(key);`

  Constructs a default key with a value of 0.

  Parameters
  key

- `ccFontKey(TCHAR charCode, c_Int32 instance);`

  Constructs a key with the specified character code and instance number.

  Requires that the instance be less than or equal to *ckMaxUInt16* + 1

  Parameters

charCode	The character code.
instance	The instance.

Notes

The default copy constructor, assign operator, and destructor are used.

Public Member Functions

key

```
c_Int32 key() const;
void key(c_Int32 k);
```

- `c_Int32 key() const;`

  Gets key value.

- `void key(c_Int32 k);`

  Sets key value.

  Parameters

  k The new key value.

charCode

```
TCHAR charCode() const;
```

Gets the 16-bit Unicode code point.

instance

```
c_Int32 instance() const;
```

Gets the instance number.

c_Int32

```
operator c_Int32() const;
```

Casts the key to a c_Int32, which is the same as the value returned by **key()**.

TCHAR

```
operator TCHAR() const;
```

Casts the key to a TCHAR, which is the same as the value returned by **charCode()**.

<

```
bool operator<(const ccFontKey& rhs) const;
```

Compares *this to rhs, ordered first by character code in increasing order and then by instance in increasing order.

ccFontKey

ccFrameAverageBuffer

```
#include <ch_cvl/frameavg.h>

template <class imgPel, class accPel> class ccFrameAverageBuffer;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

ccFrameAverageBuffer is a template class that provides functionality to accumulate images with pixels of type *imgPel* into a buffer with pixels of type *accPel*, and to compute the average image. Averaged images can be cleaner images than raw camera images, and can be presented to other CVL vision tools to produce better results.

imgPel — Template parameter specifying the pel type of the images being accumulated. For example, **c_UInt8**.

accPel — Template parameter specifying the pel type of the accumulated image. For example, **c_UInt16**.

For example, if you are accumulating 8-bit images into a 16-bit accumulator, *imgPel* is **c_UInt8** and *accPel* is **c_UInt16**. If you are accumulating 12-bit images into a 16-bit accumulator, both *imgPel* and *accPel* would be **c_UInt16**.

Note The **ccFrameAverageBuffer** class is currently implemented only for the case where *imgPel* = **c_UInt8** and *accPel* = **c_UInt16**.

ccFrameAverageBuffer operates in two modes, standard and rolling average. In standard mode, you can accumulate **ccPelBuffer**s, find their average, and optionally find the standard deviation at each pixel. In rolling average mode, this class maintains a rolling average of the last N images.

Constructors/Destructors

ccFrameAverageBuffer

```
ccFrameAverageBuffer();

ccFrameAverageBuffer(const ccFrameAverageBuffer
  <imgPel, accPel> &buffer);
```

- `ccFrameAverageBuffer();`

  Default constructor. Creates a new frame average buffer in standard mode. The size of internal buffers is determined by the first image added with **add()**. **accumulateStats()** is set to false, so statistics are not accumulated.

- `ccFrameAverageBuffer(const ccFrameAverageBuffer <imgPel, accPel> &buffer);`

  Copy constructor.

Operators

operator==

```
bool operator== (ccFrameAverageBuffer<imgPel, accPel>
  &buffer);
```

Returns true if the current frame buffer average is identical to *buffer*.

operator=

```
ccFrameAverageBuffer<imgPel, accPel>& operator=
  (const ccFrameAverageBuffer &buffer);
```

Copies the accumulator and all its pixels to a new pel root.

Public Member Functions

setStandardMode

```
void setStandardMode(bool accumulateStats=false);
```

Sets the frame averaging mode to standard mode (**ccFrameAverageDefs::eStandardMode**). The current accumulation buffer is reset.

Parameters

accumulateStats

If true, statistics are accumulated that allow the computation of a standard deviation image. If false, no statistics are accumulated.

setRollingAverageMode

```
void setRollingAverageMode(c_Int32 maxNumRollingFrames);
```

Sets the frame averaging mode to rolling average mode (**ccFrameAverageDefs::eRollingAverageMode**).

Notes

Rolling average mode is currently only implemented for **ccFrameAverageBuffer** instances where the *imgPel* type is **c_UInt8** and the *accPel* type is **c_UInt16**. Any other values makes **setRollingAverageMode()** return *NotImplemented.*

Parameters

maxNumRollingFrames

The number of frames to use in the rolling average computation. Must be a power of two between 2 and 256, inclusive.

Throws

ccFrameAverageDefs::NotImplemented

If *maxNumRollingFrames* is not a power of two or is > 256.

NotImplemented is also thrown if the *imgPel* type of the current **ccFrameAverageBuffer** instance is not **c_UInt8**, or if the *accPel* type is not **c_UInt16**.

ccFrameAverageDefs::BadParams

If *maxNumRollingFrames* is <= 0.

averageMode

```
ccFrameAverageDefs::AveragingMode averageMode() const;
```

Returns the current averaging mode setting for the current frame average buffer.

reset

```
void reset();
```

Resets the current frame average buffer to have an empty accumulator, but does not alter the current frame averaging mode or statistics accumulation mode.

maxNumRollingFrames

```
c_Int32 maxNumRollingFrames() const;
```

Returns the number of images currently set for computing the rolling average when in rolling average mode.

Throws

ccFrameAverageDefs::BadParams

If the current buffer is in standard mode.

accumulateStats

```
bool accumulateStats() const;
void accumulateStats(bool);
```

- `bool accumulateStats() const;`

  Returns true if the current frame average buffer is set to accumulate standard deviation statistics; returns false if not.

- `void accumulateStats(bool);`

  Setting **accumulateStats(true)** tells the current frame average buffer to accumulate the necessary statistics to be able to compute a standard deviation image from the accumulated images. Setting **accumulateStats(false)** disables the accumulation of standard deviation statistics. Setting either true or false always calls **reset()** first, even if the current state of the buffer is the same.

  Throws

  ccFrameAverageDefs::NotImplemented
  If the current buffer is in rolling average mode.

hasStdDevImage

```
bool hasStdDevImage() const;
```

Returns true if a standard deviation image is available to be retrieved. That is, it returns true if **accumulateStats()** is true and **numFrames()** > 1.

Throws

ccFrameAverageDefs::NotImplemented
If the current buffer is in rolling average mode.

stdDevImage

```
ccPelBuffer_const<imgPel> stdDevImage() const;
```

If **hasStdDevImage()** is true, returns the standard deviation image.

Throws

ccFrameAverageDefs::NotImplemented
If the current buffer is in rolling average mode.

ccFrameAverageDefs::BadParams
If **hasStdDevImage()** is false, or if **accumulateStats()** is false.

add

```
void add (ccPelBuffer_const <imgPel> &pelbuf);
```

Adds the pixel values of *pelbuf* to the accumulator image. If the accumulator image is unbound, copies *pelbuf* to the accumulator image.

Parameters

pelbuf The pel buffer to copy to the accumulator. The template parameter *imgPel* must be the same template parameter you specifed as the first template parameter of your **ccFrameAverageBuffer** instance, probably **c_UInt8**.

Throws

ccFrameAverageDefs::ImageUnbound
If *pelbuf* is unbound.

ccFrameAverageDefs::WindowsDiffer
If *pelbuf* 's region is not identical to the region of the current frame average bufer.

ccFrameAverageDefs::NotImplemented
If in rolling average mode and the *imgPel* type of the current **ccFrameAverageBuffer** instance is not **c_UInt8**, or the *accPel* type is not **c_UInt16**.

numFrames

```
c_Int32 numFrames() const;
```

Returns the number of pel buffer frames that have been accumulated so far.

Notes

When in rolling average mode, **numFrames()** returns the number of frames included in the frame average buffer, until a maximum value equal to the number of images used in averaging is reached. This is the *maxNumRollingFrames* value set with **setRollingAverageMode()**.

average

```
ccPelBuffer<imgPel> average() const;

void average(ccPelBuffer<imgPel> &destImage) const;
```

- `ccPelBuffer<imgPel> average() const;`

  Computes the frame-averaged result by dividing all pel values in the average frame buffer by the number of frames accumulated, rounding if necessary, and returns the averaged pel buffer. *imgPel* must be the same template parameter you specifed as the first template parameter of your **ccFrameAverageBuffer** instance, probably **c_UInt8**.

- `void average(ccPelBuffer<imgPel> &destImage) const;`

  Computes the frame-averaged result as above, but places the result in *destImage. imgPel* must be the same template parameter you specifed as the first template parameter of your **ccFrameAverageBuffer** instance, probably **c_UInt8**.

Notes

If this current frame average buffer's accumulator image is unbound, the result of this function is an unbound pel buffer.

If *destImage* is specified, **average()** operates only on the intersection of the image coordinates of *destImage* and the current frame average buffer, and put the averaging result into *destImage.*

If *destImage* is unbound, **average()** sets it to have the same height, width, and origin as the current frame average buffer, and puts the results of the average operation into it.

Throws

ccFrameAverageDefs::RollingBufferNotFull

For rolling average mode, if **numFrames()** is less than the number of images you specifed to be used in computing the rolling average (*maxNumRollingFrames*).

ccFrameAverageDefs::DivideByZero

If the current frame average buffer's accumulator image is bound, but the number of images accumulated so far is zero.

ccFrameAverageDefs::NotImplemented

If in rolling average mode and the *imgPel* type of the current **ccFrameAverageBuffer** instance is not **c_UInt8**, or the *accPel* type is not **c_UInt16**.

ccFrameAverageDefs

```
#include <ch_cvl/frameavg.h>

class ccFrameAverageDefs;
```

A name space that holds enumerations used with the image averaging tools.

Enumerations

AveragingMode

```
enum AveragingMode {eStandardMode=0,
                    eRollingAverageMode};
```

ccFrameAverageDefs

ccFrameGrabber

```
#include <ch_cvl/fg.h>

class ccFrameGrabber;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This abstract class is used to describe Cognex frame grabbers. Your application will typically use a class derived from this class to specify a Cognex frame grabber.

Constructors/Destructors

Construction and destruction of derived classes is done automatically.

Enumerations

enum

```
enum {ckCvmNotPresent = -1, ckNoCvmConnector = -2};
```

Possible return values for **cvmId()**.

Value	Meaning
ckCvmNotPresent = -1	The frame grabber does support CVMs but none is currently attached.
ckNoCvmConnector = -2	The frame grabber does not support CVMs.

Public Member Functions

numChannels

```
c_Int32 numChannels() const;
```

Returns the number of video channels supported by this frame grabber. In general, this is the number of simultaneous acquisitions that this frame grabber can perform.

Throws

ccBoard::HardwareInUse

If the frame grabber is being used by another process. Recovery might be possible by waiting for the other process to exit, then calling this function again.

Throws other exceptions derived from **ccException** if a fatal error occurs. No recovery is possible, other than to display **ccException::message()** and exit gracefully.

numCameraPort

```
c_Int32 numCameraPort(const ccVideoFormat& fmt) const;
```

Parameters

fmt A video format.

Returns the number of camera ports that can be used with the given video format. Zero is returned if the video format is not supported on this frame grabber.

Throws

ccBoard::HardwareInUse

If the frame grabber is being used by another process. Recovery might be possible by waiting for the other process to exit, then calling this function again.

Throws other exceptions derived from **ccException** if a fatal error occurs. No recovery is possible, other than to display **ccException::message()** and exit gracefully.

isSupportedEx

```
bool isSupportedEx (
  const ccVideoFormat& fmt,
  ceImageFormat imgFmt=ceImageFormat_Unknown) const;
```

Returns true if the frame grabber supports the given video format. Otherwise returns false.

Parameters

ccVideoFormat A video format.

imgFmt An image format.

Notes

If a specific *imgFmt* is specified, true is returned only if images of the specified type can be acquired. If *ceImageFormat_Unknown* is specified, it is interpreted as a wildcard value, and true is returned if any image format is supported.

Throws

ccBoard::HardwareInUse

If the frame grabber is being used by another process. Recovery might be possible by waiting for the other process to exit, then calling this function again.

Throws other exceptions derived from **ccException** if a fatal error occurs. No recovery is possible, other than to display **ccException::message()** and exit gracefully.

isSupported

```
bool isSupported(const ccVideoFormat& fmt) const;
```

Parameters

ccVideoFormat A video format.

Returns *true* if the frame grabber supports the given video format. Otherwise returns *false*.

Notes

This is a legacy version of **isSupportedEX** (above) that will be deprecated in a future release. It returns true if a **ccGreyAcqFifo** can be constructed to acquire images using the specified video format.

Throws

ccBoard::HardwareInUse

If the frame grabber is being used by another process. Recovery might be possible by waiting for the other process to exit, then calling this function again.

Throws other exceptions derived from **ccException** if a fatal error occurs. No recovery is possible, other than to display **ccException::message()** and exit gracefully.

supportsCCFs

```
bool supportsCCFs() const;
```

Returns true if the frame grabber supports video formats that are formed from Cognex Configuration Files (CCFs).

Notes

The frame grabber might not necessarily support all CCF video formats. This function indicates whether the frame grabber has the ability to support *any* CCF formats.

Throws

ccBoard::HardwareInUse
If the frame grabber is being used by another process. Recovery might be possible by waiting for the other process to exit, then calling this function again.

Throws other exceptions derived from **ccException** if a fatal error occurs. No recovery is possible, other than to display **ccException::message()** and exit gracefully.

cvmId

```
c_Int32 cvmId() const;
```

Returns the numerical ID of the CVM attached to this frame grabber. Or, returns *ckNoCvmConnector* if the frame grabber does not support CVMs, or returns *ckCvmNotPresent* if the frame grabber does support CVMs but no CVM is currently attached.

Notes

cvmId() should be used for information purposes only and not as a substitute for other query functions, such as **ccFrameGrabber::numCameraPort()** or **ccTriggerProp::couldSlaveTo()**.

Throws

ccBoard::HardwareInUse
If the frame grabber is being used by another process. Recovery might be possible by waiting for the other process to exit, then calling this function again.

Throws other exceptions derived from **ccException** if a fatal error occurs. No recovery is possible, other than to display **ccException::message()** and exit gracefully.

Static Functions

count

```
static c_Int32 count();
```

Returns the number of frame grabbers installed.

Notes

This function is not available at static initialization time

get

```
static ccFrameGrabber& get(c_Int32 i = 0);
```

Return the *i*th frame grabber installed. Frame grabber numbering starts at 0.

Parameters

i The frame grabber number.

Throws

ccBoard::BadParams

i is not in the range 0 through **count()** - 1, or no frame grabbers are installed (**count()** returns 0).

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding

The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse

The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error, a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8100m**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occurred while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from

an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

ccGaussSampleParams

```
#include <ch_cvl/gaussmpl.h>

class ccGaussSampleParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the parameters you specify to control the Gaussian Sampling tool.

Constructors/Destructors

ccGaussSampleParams

```
ccGaussSampleParams (const ccIPair& samples,
  const ccIPair& smoothnesses, double scale = 1.0,
  bool canUsePelRoot = true);

ccGaussSampleParams (c_Int32 sample = 1,
  c_Int32 smoothness = 2, double scale = 1.0,
  bool canUsePelRoot = true);
```

- ```
  ccGaussSampleParams (const ccIPair& samples,
    const ccIPair& smoothnesses, double scale = 1.0,
    bool canUsePelRoot = true);
  ```

  Constructs a **ccGaussSampleParams** using the specified values. This constructor lets you specify independent values for x-axis and y-axis smoothing and sampling values.

  Parameters

  samples — A **ccIPair** containing the x-axis and y-axis sampling factors. The output image produced by **cfGaussSample()** is reduced in size by the values specified in *samples*.

  smoothnesses — A **ccIPair** containing the x-axis and y-axis smoothing values. These values are in image coordinate system units.

  scale — The output scaling factor to apply during the smoothing operation.
  ```

canUsePelRoot If true, **cfGaussSample()** will use any available pixels present in the input image's root image when computing the smoothed image. If false, **cfGaussSample()** only uses pixels within the input image.

- ```
  ccGaussSampleParams (c_Int32 sample = 1,
    c_Int32 smoothness = 2, double scale = 1.0,
    bool canUsePelRoot = true);
  ```

  Constructs a **ccGaussSampleParams** using the specified values. This constructor accepts a single value for both x- and y-axis smoothing and a single value for both x- and y-axis sampling.

  Parameters

  sample The sampling factor. The output image produced by **cfGaussSample()** is reduced in size by the *sample*.

  smoothness The smoothing value in image coordinate system units

  scale The output scaling factor to apply during the smoothing operation

  canUsePelRoot If true, **cfGaussSample()** will use any available pixels present in the input image's root image when computing the smoothed image. If false, **cfGaussSample()** only uses pixels within the input image.

Public Member Functions

sample

```
ccIPair sample() const;

void sample(const ccIPair& samplings);

void sample(c_Int32 sampling);
```

- ```
  ccIPair sample() const;
  ```

  Returns the x- and y-axis sampling rates

- ```
  void sample(const ccIPair& samplingRates);
  ```

  Sets the x-axis and y-axis sampling rates.

  Parameters

  samplingRates The sampling values.

  Throws
  ```

ccGaussSampleParams::BadParams
samplingRates.x() or *samplingRates.y()* is less than or equal to 0.

Notes

The client coordinate system of the image produced by **cfGaussSample()** is updated to reflect the effect of the sampling rate.

- `void sample(c_Int32 sampling);`

Sets the sampling rate.

Parameters

sampling This value for both x- and y-axis sampling

Throws

ccGaussSampleParams::BadParams
sampling is less than or equal to 0.

Notes

The client coordinate system of the image produced by **cfGaussSample()** is updated to reflect the effect of the sampling rate.

smoothness

```
ccIPair smoothness () const;
void smoothness (const ccIPair& smoothnesses);
void smoothness (c_Int32 smoothness);
```

- `ccIPair smoothness () const;`

Returns the x- and y-axis smoothness values.

- `void smoothness (const ccIPair& smoothnesses);`

Sets separate smoothness values for the x- and y-axis directions. The smoothness value determines the sigma of the Gaussian curve and in turn the size of the Gaussian kernel. Specifying a smoothness value greater than *ccGaussSampleParams::kMaxSmoothness* will result in slower execution times than smaller values.

Parameters

smoothnesses The x- and y-axis smoothness values to set

Throws

ccGaussSampleParams::BadParams
Either *smoothnesses.x()* or *smoothnesses.y()* is less than 0.

- `void smoothness (c_Int32 smoothness);`

  Sets a single smoothness value for both the x- and y-axis directions. The smoothness value determines the sigma of the Gaussian curve and in turn the size of the Gaussian kernel. Specifying a smoothness value greater than *ccGaussSampleParams::kMaxSmoothness* will result in slower execution times than smaller values.

  Parameters

  smoothness The smoothing value to set

  Throws

  ccGaussSampleParams::BadParams
  smoothness is less than 0.

---

scale

`double scale () const;`

`void scale (double scale);`

- `double scale () const;`

  Returns the scaling value applied during the smoothing operation.

- `void scale (double scale);`

  Sets the scaling value applied during the smoothing operation. A scaling value of 1.0 applied to an 8-bit input image produces values in the range 0-255. A scaling value of greater than 1.0 can improve the dynamic range if the input image has low contrast, but it might cause overflow.

  Parameters

  scale The scaling factor to set.

  Throws

  ccGaussSampleParams::BadParams
  scale is 0.0 or less.

sigma

`ccDPair sigma () const;`

Returns standard deviation of the Gaussian curve in both the x- and y-axis. The standard deviation is computed using the following formula:

$$\sigma = \frac{\sqrt{s(s+2)}}{2}$$

where *s* is the smoothing value.

canUsePelRoot

```
bool canUsePelRoot () const;
void canUsePelRoot (bool useRootPels);
```

- `bool canUsePelRoot () const;`

  Returns true if this **ccGaussSampleParams** is configured to use any available pixels from the root image to compute the output image, false if only those pixels inside the input image are considered.

- `void canUsePelRoot (bool useRootPels);`

  Controls whether this **ccGaussSampleParams** uses any available pixels from the root image to compute the output image.

  Parameters

  useRootPels — If true, the **ccGaussSampleParams** uses root pixels; if false, it does not

ccGaussSampleParams

ccGenAnnulus

```
#include <ch_cvl/shapes.h>

class ccGenAnnulus : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a generalized annulus, a shape made up of two generalized rectangles (**ccGenRect**).

Constructors/Destructors

ccGenAnnulus

```
ccGenAnnulus();

ccGenAnnulus(const cc2Vect& center,
  const cc2Vect& radius1, const cc2Vect& radius2,
  const ccRadian orient, const cc2Vect& round);

ccGenAnnulus(const cc2Vect& center,
  const cc2Vect& radius1, const cc2Vect& radius2,
  const ccRadian orient,
  const cc2Vect& round1, const cc2Vect& round2);

ccGenAnnulus(const ccGenRect& r1, const ccGenRect& r2);

ccGenAnnulus(const ccAnnulus& a);
```

- `ccGenAnnulus();`

  Default constructor. Constructs a degenerate generalized annulus.

  Notes

  The default constructor is provided to support arrays and STL containers. Do not try to examine or use a default-constructed **ccGenAnnulus**.

- ```
  ccGenAnnulus(const cc2Vect& center,
    const cc2Vect& radius1,const cc2Vect& radius2,
    const ccRadian orient, const cc2Vect& round);
  ```

  Creates a generalized annulus. Both generalized rectangles have the same roundness.

  Parameters

center	The center of the annulus.
radius1	The radii of the outer generalized rectangle.
radius2	The radii of the inner generalized rectangle.
orient	The orientation of the annulus.
round	The roundness of both generalized rectangles.

  Throws

  ccShapesError::BadRadius
  One of the radii is less than zero.

  ccShapesError::NotConcentric
  radius1 and *radius2* do not have the same center.

  Notes

  The components of *round* for each **ccGenRect** are internally clipped so that they do not exceed the corresponding components of *radii* for that **ccGenRect**.

- ```
  ccGenAnnulus(const cc2Vect& center,
    const cc2Vect& radius1,const cc2Vect& radius2,
    const ccRadian orient,
    const cc2Vect& round1, const cc2Vect& round2);
  ```

  Creates a generalized annulus. Each generalized rectangle has its own roundness.

  Parameters

center	The center of the annulus.
radius1	The radii of the outer generalized rectangle.
radius2	The radii of the inner generalized rectangle.
orient	The orientation of the annulus.
round1	The roundness of the outer generalized rectangle.
round2	The roundness of the inner generalized rectangle.

Throws

ccShapesError::BadRadius
One of the radii is less than zero.

ccShapesError::NotConcentric
radius1 and *radius2* do not have the same center.

Notes

The components of *round1* and *round2* are internally clipped, if necessary, so that they do not exceed the corresponding components of *radii1* and *radii2*, respectively.

- `ccGenAnnulus(const ccGenRect& r1, const ccGenRect& r2);`

Creates a generalized annulus from two generalized rectangles.

Parameters

r1 The generalized rectangle.

r2 The generalized rectangle.

Throws

ccShapesError::BadRadius
One of the radii is less than zero.

ccShapesError::NotConcentric
r1 and *r2* do not have the same center.

- `ccGenAnnulus(const ccAnnulus& a);`

Conversion constructor. Constructs a **ccGenAnnulus** from the given **ccAnnulus**.

Parameters

a The **ccAnnulus**.

Throws

ccShapesError::BadRadius
Either of *a*'s radii is less than or equal to zero.

ccShapesError::NotConcentric
a's radii do not have the same center.

Operators

operator== `bool operator==(const ccGenAnnulus& that) const;`

Returns true if this generalized annulus is equal to that, false otherwise.

Parameters

that The other generalized annulus.

operator!=

```
bool operator!=(const ccGenAnnulus& that) const;
```

Returns true if this generalized annulus is not equal to that, false otherwise.

Parameters

that The other generalized annulus.

Public Member Functions

center

```
ccPoint center () const;
```

Returns the center of the generalized annulus.

innerRadius

```
cc2Vect innerRadius() const;
```

Returns the radii of the inner generalized rectangle.

Notes

The **innerRadius()** and **outerRadius()** of the generalized annulus are silently swapped if the **innerRadius()** is outside the **outerRadius()**.

outerRadius

```
cc2Vect outerRadius() const;
```

Returns the radii of the outer generalized rectangle.

Notes

The **innerRadius()** and **outerRadius()** of the generalized annulus are silently swapped if the **innerRadius()** is outside the **outerRadius()**.

orient

```
ccRadian orient () const;
```

Returns the orientation of the generalized annulus.

round

```
cc2Vect round () const;
```

Returns the roundness of the inner generalized rectangle.

innerRound

```
cc2Vect innerRound() const;
```

Returns the roundness of the inner generalized rectangle.

outerRound

```
cc2Vect outerRound() const;
```

Returns the roundness of the outer generalized rectangle.

innerGenRect

```
const ccGenRect& innerGenRect() const;
```

Returns the inner generalized rectangle.

outerGenRect

```
const ccGenRect& outerGenRect() const;
```

Returns the inner generalized rectangle.

map

```
ccGenAnnulus map(const cc2Xform& c) const;
```

Returns a generalized annulus that is the result of mapping this generalized annulus with the transformation object *c*.

Parameters

c	The transformation object.

Notes

The transform *c* must be singular. If it is not, use **mapShape()** instead.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this generalized annulus.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For generalized annuli, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For generalized annuli, this function always returns true, including generalized annuli that are degenerate. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For generalized annuli, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For generalized annuli, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

For generalized annuli, this function returns true if **hasTangent()** is true for either the inner or outer generalized rectangle. It returns false if **hasTangent()** is false for both the inner and outer rectangles. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For generalized annuli, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible `virtual bool isReversible() const;`

For generalized annuli, this function always returns false. See **ccShape::reverse()** for more information.

isRightHanded `virtual bool isRightHanded() const;`

Returns true if this generalized annulus is right-handed. See **ccShape::isRightHanded()** for more information.

boundingBox `virtual ccRect boundingBox() const;`

Returns the smallest rectangle that encloses this generalized annulus. See **ccShape::boundingBox()** for more information.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const;`

Returns the nearest point on the boundary of this generalized annulus to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this generalized annulus to the given point, as determined by **nearestPoint()**.

Parameters

info	Shape information for this generalized annulus.
point	The point.

See **ccShape::nearestPerimPos()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores generalized annuli for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this generalized annulus mapped by *X*.

Parameters

X	The transformation object.

Notes

If the transform *X* is singular, this function returns a **ccRegionTree**. Otherwise it returns a **ccGenAnnulus**.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccLineSeg**s and, if the underlying **ccGenRect**s have rounded corners, **ccEllipseArcs**. See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect&) const;
```

Returns true if the given point is within this generalized annulus.

Parameters

p The point.

Notes

This function returns false if either of the underlying rectangles is degenerate.

See **ccShape::within()** for more information.

subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this generalized annulus over the given perimeter range.

Parameters

info Shape information for this shape.

range The perimeter range.

See **ccShape::subShape()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

ccGeneralShapeTree

```
#include <ch_cvl/shaptree.h>

class ccGeneralShapeTree : public ccShapeTree;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccGeneralShapeTree** class is a concrete class for maintaining arbitrary hierarchies of shapes. There are no constraints on the number or types of children any node in the hierarchy may have.

Constructors/Destructors

ccGeneralShapeTree

```
ccGeneralShapeTree();
```

Default constructor. Constructs a **ccGeneralShapeTree** containing no children.

Copies and assignments are shallow, that is, they are achieved by copying pointer handles. Hence, these operations lead to sharing of children between different **ccShapeTree**s.

Public Member Functions

connect

```
ccGeneralShapeTreePtrh connect(const double tolerance)
  const;
```

Connects the immediate children of this general shape tree into longer contours where possible. The returned tree contains the same children as this tree, with the following modifications:

- General shape tree children of this general shape tree are recursively connected.

- Any connectable open contour children of this tree are connected into contour trees, which become initial children of the result tree. These contour trees will contain as many children as can be connected into a single chain, and may be open or closed depending on the topology of the connections. Thus, the open contour children of this tree may appear either as direct children of the result tree, or within new contour trees that are direct children of the result tree.
- All non-connectable children (closed contours) of the original tree appear last among the children of the returned tree.

Two open contour shapes are connectable if the distance between two of their end points (one from each contour) is less than the specified tolerance. Each end point may be connected to at most one other end point. If end points from a number of different contours are mutually within the specified tolerance from each other, the algorithm connects the closest end points first. An open contour may be connected to itself if the distance between its start point and end point is less than the tolerance.

The connection algorithm automatically checks all combinations of end points for possible connections. For example, there are four possible ways to connect two line segments, as either the start or end point of each segment may be chosen for the connection. In constructing contour trees from the connected shapes, the component shapes are reversed when necessary to maintain the contour tree constraint that the end point of child *i* is implicitly connected to the start point of child *i*+1.

There are two choices for the direction of the final connected contour. The algorithm chooses whichever direction requires fewer reversals of individual component shapes.

Parameters

tolerance	The maximum distance between endpoints of different open contours that can be connected to each other.

Notes

The **connect()** method requires two or more open contours in order to build a connected, closed contour. It will not connect the endpoints of a single closed contour.

This method recursively processes an entire shape hierarchy. However, only open-contour shapes that have the same general shape tree parent are candidates for connection. Thus, general shape tree nodes can be used to partition the connection candidates. In some applications, it may be useful to call **flatten()** before calling **connect()**.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this general shape tree.

isOpenContour `virtual bool isOpenContour() const;`

For general shape trees, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion `virtual bool isRegion() const;`

For general shape trees, this function always returns false. See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

Returns true if all primitive shapes in the hierarchy are finite. See **ccShape::isFinite()** for more information.

isReversible `virtual bool isReversible() const;`

Returns true if and only if all descendants of this **ccGeneralShapeTree** are reversible. See **ccShape::isReversible()** for more information.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const;`

Returns the nearest point on the boundary of this general shape tree to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p The point.

Notes

The nearest point to *p* is taken among all primitives in the hierarchy.

See **ccShape::nearestPoint()** for more information.

reverse `virtual ccShapePtrh reverse() const;`

Returns the reversed version of this general shape tree.

Notes

This method reverses the individual primitives of the hierarchy and their order within the hierarchy.

See **ccShape::reverse()** for more information.

sample

```
virtual void sample(const ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this general shape tree.

Parameters

params Parameters object specifying details of how the sampling should be done.

result Result object to which position and tangent chains are stored.

Notes

A separate chain is generated for each primitive in the hierarchy.

See **ccShape::sample()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this general shape tree over the given perimeter range.

Parameters

info Shape information for this general shape tree.

range The perimeter range.

See **ccShape::subShape()** for more information.

ccGenPoly

```
#include <ch_cvl/genpoly.h>

class ccGenPoly : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Not intended
Archiveable	Complex

The **ccGenPoly** class encapsulates a general polygon shape consisting of sequentially connected 2D arc and line segments connected at vertices. Each vertex is specified by a point and a rounding size. The vertex point specifies the coordinate of the vertex. The rounding size specifies the circular radius of rounding of the vertex. The general polygon object is created incrementally by adding vertices one at a time, and adjusting the curvature of the resulting segments for rounding.

See the *Shapes* chapter in the *CVL User's Guide* for more information.

Constructors/Destructors

ccGenPoly

```
ccGenPoly();
ccGenPoly(const ccLineSeg& ls);
ccGenPoly(const ccRect& rect);
ccGenPoly(const ccCircle& circle);
ccGenPoly(const ccEllipse2& ellipse);
ccGenPoly(const ccEllipseArc2& ellArc);
ccGenPoly(const ccGenRect& gr);
ccGenPoly(const ccEllipseAnnulusSection& eas);
ccGenPoly(const ccGenPoly & gp);
virtual ~ccGenPoly();
```

- `ccGenPoly();`

  Constructs a general polygon object with no vertices or segments.

- `ccGenPoly(const ccLineSeg& ls);`

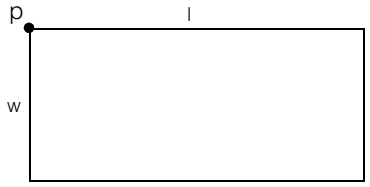
  Conversion constructor. Constructs a mutable **ccGenPoly** object based on an object of type **ccLineSeg**, a straight line between two points. See the following example:

  Parameters

  ls An object of type **ccLineSeg**.

- `ccGenPoly(const ccRect& rect);`

  Conversion constructor. Constructs a mutable **ccGenPoly** object based on an object of type **ccRect**, a rectangle at point p, of width w, and length l. See the following example:

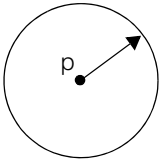


  Parameters

  rect An object of type **ccRect**.

- `ccGenPoly(const ccCircle& circle);`

  Conversion constructor. Constructs a mutable **ccGenPoly** object based on an object of type **ccCircle**, a circle at point p with a radius r. See the following example:

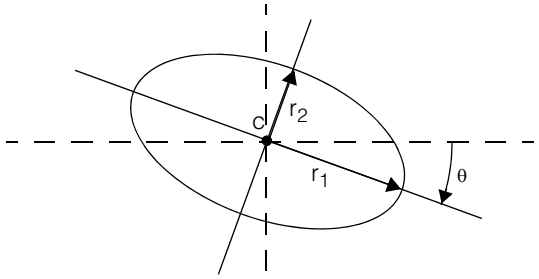


  Parametersh

  circle An object of type **ccCircle**.

- `ccGenPoly(const ccEllipse2& ellipse);`

  Conversion constructor. Constructs an immutable **ccGenPoly** object based on an object of type **ccEllipse2**, an ellipse as shown in the following example:

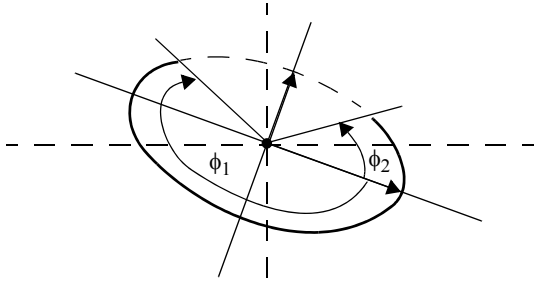


  Parameters

  ellipse An object of type **ccEllipse2**.

- `ccGenPoly(const ccEllipseArc2& ellArc);`

  Conversion constructor. Constructs an immutable **ccGenPoly** object based on an object of type **ccEllipseArc2**. An ellipse arc is a partial ellipse with the end points specified by the angles ϕ_1 and ϕ_2. See the following example:

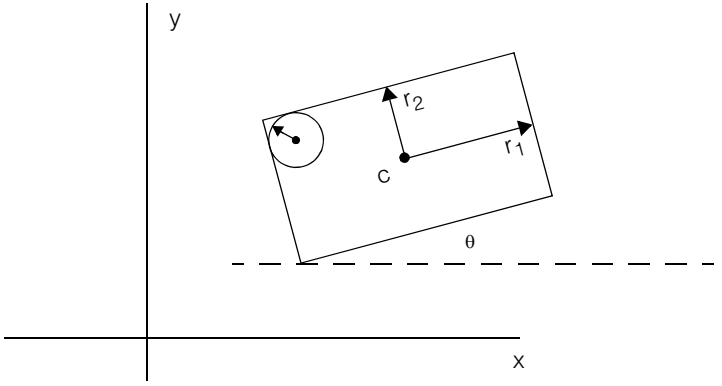


  Parameters

  ellArc An object of type **ccEllipseArc2**.

- `ccGenPoly(const ccGenRect& gr);`

  Conversion constructor. Constructs an immutable **ccGenPoly** object based on an object of type **ccGenRect**, a general rectangle at angle θ in x,y space. The rectangle is defined by its center point (c) and r_1 and r_2 which are half the length and half the width respectively. See the following example:

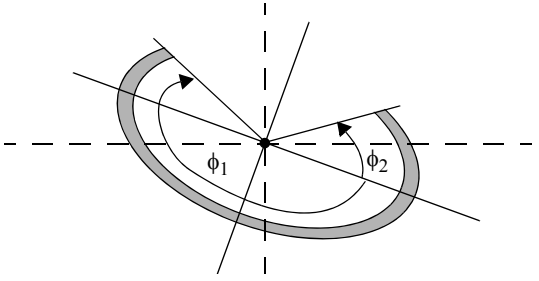


  Parameters

  gr An object of type **ccGenRect**.

- `ccGenPoly(const ccEllipseAnnulusSection& eas);`

  Conversion constructor. Constructs an immutable **ccGenPoly** object based on an object of type **ccEllipseAnnulusSection**, a partial ellipse annulus with the end points specified by the angles ϕ_1 and ϕ_2. See the following example:



  Parameters

  eas An object of type **ccEllipseAnnulusSection**.

- `ccGenPoly(const ccGenPoly & gp);`

  Copy constructor. If *gp* is immutable, the constructed copy will also be immutable. If *gp* is mutable, the constructed copy will also be mutable.

Parameters

gp The **ccGenPoly** object to copy.

- `virtual ~ccGenPoly();`

  Destructor. Frees memory and deletes the object.

Operators

operator= `ccGenPoly & operator=(const ccGenPoly & gp);`

Assignment operator. If *gp* is immutable, the assigned object will also be immutable. If *gp* is mutable, the assigned object will also be mutable.

Parameters

gp The assignment source object.

operator== `bool operator==(const ccGenPoly& other) const;`

This operator allows you to compare two **ccGenPoly** objects. For example,

```
if(poly1 == poly2) {.....}
```

The equality expression evaluates to *true* if the objects are equal. It is *false* if they are not equal. Two general polygons are considered equal only if vertex and segment information is the same to within reasonable floating point precision.

Notes

Immutable general polygons that have been mapped by different sequences of linear transformations will in general not be considered equal unless they are equal before they are mapped.

Parameters

other The right-hand **ccGenPoly** object. For example, *`poly2`*.

operator!= `bool operator!=(const ccGenPoly& other) const;`

This operator allows you to compare two **ccGenPoly** objects. For example,

```
if(poly1 != poly2) {.....}
```

The inequality expression evaluates to *true* if the objects are not equal. It is *false* if they are equal. Two general polygons are considered equal only if vertex and segment information is the same to within reasonable floating point precision.

Notes

Immutable general polygons that have been mapped by different sequences of linear transformations will in general not be considered equal unless they are equal before they are mapped.

Parameters

other The right-hand **ccGenPoly** object. For example, *poly2*.

Public Member Functions

convert

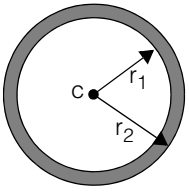
```
static cmStd vector<ccGenPoly> convert(
  const ccAnnulus& annulus);

static cmStd vector<ccGenPoly> convert(
  const ccGenAnnulus& gen_annulus);

static cmStd vector<ccGenPoly> convert(
  const ccEllipseAnnulus& ellipse_annulus);
```

- ```
  static cmStd vector<ccGenPoly> convert(
    const ccAnnulus& annulus);
  ```

Returns an equivalent pair of mutable general polygon objects for the annulus provided. The routine returns one **ccGenPoly** for the inside circle, and one for the outside circle. See the example annulus below:

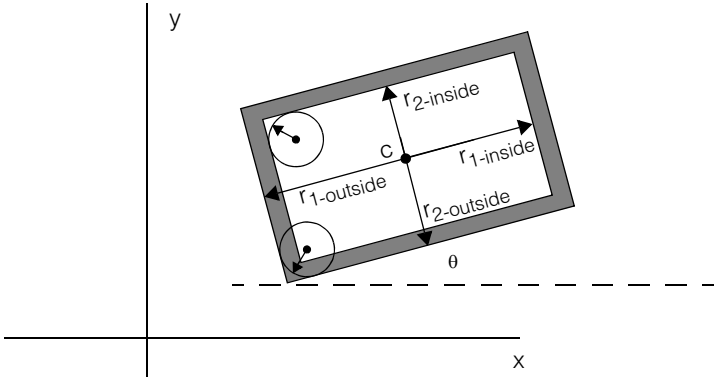


Parameters

annulus The annulus object to convert.
  ```

- `static cmStd vector<ccGenPoly> convert(`
  `const ccGenAnnulus& gen_annulus);`

  Returns an equivalent pair of mutable general polygon objects for the general annulus provided. The routine returns one **ccGenPoly** for the inside rectangle, and one for the outside rectangle. See the example below:

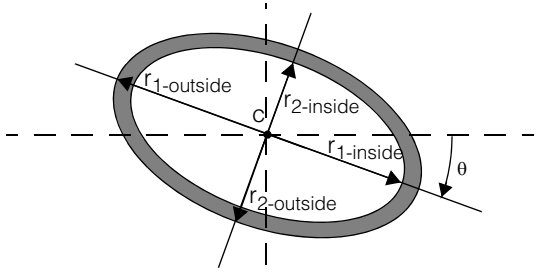


Parameters

gen_annulus The general annulus object to convert.

- `static cmStd vector<ccGenPoly> convert(`
  `const ccEllipseAnnulus& ellipse_annulus);`

  Returns an equivalent pair of mutable general polygon objects for the ellipse annulus provided. The routine returns one **ccGenPoly** for the inside ellipse, and one for the outside ellipse. See the example below:



Parameters

ellipse_annulus The ellipse annulus object to convert.

isMutable `bool isMutable() const;`

Returns *true* if this object is mutable; returns *false* otherwise.

numVertices

```
c_Int32 numVertices() const;
```

Returns the number of vertices in the general polygon contained in this object.

numSegments

```
c_Int32 numSegments() const;
```

Returns the number of segments in the general polygon contained in this object.

isOpen

```
bool isOpen() const;
```

Returns *true* if the general polygon in this object is open; returns *false* if the general polygon is closed.

degen

```
bool degen() const;
```

Returns *true* if the general polygon in this object is degenerate (contains fewer than 2 vertices). Returns *false* otherwise.

insertVertex

```
void insertVertex(
  const cc2Vect & vertexPoint,
  double roundingSize = 0.0,
  const ccRadian & previousSegmentAngleSpan =
                                              ccRadian(0.0));

void insertVertex(
  c_Int32 previousVertexIndex,
  const cc2Vect & vertexPoint,
  double roundingSize = 0.0,
  const ccRadian & previousSegmentAngleSpan =
                                              ccRadian(0.0));
```

-
```
void insertVertex(
  const cc2Vect & vertexPoint,
  double roundingSize = 0.0,
  const ccRadian & previousSegmentAngleSpan =
                                              ccRadian(0.0));
```

Inserts the specified vertex after the last vertex in the general polygon. The vertex rounding and previous segment angle span are also specified.

Parameters

vertexPoint	The x,y location of the inserted vertex.
roundingSize	The vertex rounding size.

previousSegmentAngleSpan
The previous segment angle span.

- ```
  void insertVertex(
    c_Int32 previousVertexIndex,
    const cc2Vect & vertexPoint,
    double roundingSize = 0.0,
    const ccRadian & previousSegmentAngleSpan =
                                                ccRadian(0.0));
  ```

  Inserts the specified vertex after the vertex with index *previousVertexIndex*. The vertex rounding and previous segment angle span are also specified.

  If *previousVertexIndex* = -1, the new vertex will become the first vertex.

  In all cases, vertex indices beyond the new vertex increase by 1 as a result of adding a new vertex.

  Parameters

  previousVertexIndex
  The index of the vertex prior to where the new vertex is inserted. A -1 indicates the new vertex should be the first vertex (index 0).

  vertexPoint The x,y location of the inserted vertex.

  roundingSize The vertex rounding size.

  previousSegmentAngleSpan
  The previous segment angle span.

  Throws

  ccShapesError::BadGeom
  If the resulting general polygon contains any crossing segments.

  ccShapesError::BadRadius
  If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

  ccShapesError::BadCoef
  If *previousVertexIndex* is not a valid vertex index or not -1.

  ccShapesError::BadGeom
  If **isMutable()** = *false*.

  Notes

  The general polygon is unaffected in case of any throw.
  ```

replaceVertex

```
void replaceVertex(
  c_Int32 vertexIndex,
  const cc2Vect & vertexPoint,
  double roundingSize = 0.0);

void replaceVertex(
  c_Int32 vertexIndex,
  const cc2Vect & vertexPoint,
  double roundingSize,
  const ccRadian & previousSegmentAngleSpan);
```

- ```
  void replaceVertex(
    c_Int32 vertexIndex,
    const cc2Vect & vertexPoint,
    double roundingSize = 0.0);
  ```

  Replaces the vertex with the specified index. The new segment angle span is the same as segment angle span of the replaced vertex.

  Parameters

  vertexIndex The index of the vertex to be replaced.

  vertexPoint The new vertex point.

  roundingSize The rounding size of the new vertex.

  Throws

  ccShapesError::BadGeom
  If the resulting general polygon contains any crossing segments.

  ccShapesError::BadRadius
  If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

  ccShapesError::BadCoef
  If *vertexIndex* is not a valid vertex index.

  ccShapesError::BadGeom
  If **isMutable()** = *false*.

  ccShapesError::BadGeom
  If the specified vertex is an end vertex of an open shape and the rounding size is not zero.

  Notes

  The general polygon is unaffected in case of any throw.
  ```

- ```
  void replaceVertex(
    c_Int32 vertexIndex,
    const cc2Vect & vertexPoint,
    double roundingSize,
    const ccRadian & previousSegmentAngleSpan);
  ```

  Replaces the vertex with the specified index. A new segment angle span is specified.

  Parameters

  vertexIndex The index of the vertex to be replaced.

  vertexPoint The new vertex point.

  roundingSize The rounding size of the new vertex.

  previousSegmentAngleSpan
  The angle span of the segment entering the vertex.

  Throws

  ccShapesError::BadGeom
  If the resulting general polygon contains any crossing segments.

  ccShapesError::BadRadius
  If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

  ccShapesError::BadCoef
  If *vertexIndex* is not a valid vertex index.

  ccShapesError::BadGeom
  If **isMutable()** = *false*.

  ccShapesError::BadGeom
  If the specified vertex is an end vertex of an open shape and the rounding size is not zero.

  ccShapesError::BadGeom
  If the shape is open and the *previousSegmentAngleSpan* is non-zero for the first vertex (vertex 0).

  Notes

  The general polygon is unaffected in case of any throw.
  ```

deleteVertex `void deleteVertex(c_Int32 vertexIndex);`

Removes the vertex with index *vertexIndex*. The resulting segment between the vertex with index *vertexIndex*-1 and the vertex with index *vertexIndex*+1 retains the segment angle span of the segment with index *vertexIndex*+1 modulo N (where N is the number of vertices before the delete).

The indices for all vertices with index greater than or equal to *vertexIndex* will decrease by 1 following this operation.

Parameters

vertexIndex The index of the vertex to delete.

Throws

ccShapesError::BadGeom
If **numVertices()** < 2 and the general polygon is closed.

ccShapesError::BadGeom
If the resulting general polygon contains any crossing segments.

ccShapesError::BadRadius
If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

ccShapesError::BadCoef
If *vertexIndex* is not a valid vertex index.

ccShapesError::BadGeom
If **isMutable()** = *false*.

Notes

The general polygon is unaffected in case of any throw.

open `void open();`

Opens the general polygon. Disconnects the index 0 vertex from the vertex with the largest index. There is no effect if the general polygon is already open.

Throws

ccShapesError::BadGeom
If **isMutable()** = *false*.

close

```
void close(
  const ccRadian & segmentAngleSpan = ccDegree(0));
```

Closes the general polygon. It connects the vertex with the largest index to the vertex with the smallest index using a segment with the specified *segmentAngleSpan*. There is no effect if the general polygon is already closed.

segmentAngleSpan
The segment angle span for the segment added when the polygon is closed.

Throws

ccShapesError::BadGeom
If the resulting general polygon contains any crossing segments.

ccShapesError::BadGeom
If **isMutable()** = *false*.

Notes

The general polygon is unaffected in case of any throw.

---

vertexPoint

```
cc2Vect vertexPoint(c_Int32 vertexIndex) const;

void vertexPoint(
  c_Int32 vertexIndex,
  const cc2Vect & newVertexPoint);
```

- ```
  cc2Vect vertexPoint(c_Int32 vertexIndex) const;
  ```

  Returns the vertex point for the vertex whose index is *vertexIndex*.

  Parameters

  vertexIndex The index of the vertex you wish to retrieve.

- ```
  void vertexPoint(
    c_Int32 vertexIndex,
    const cc2Vect & newVertexPoint);
  ```

  Sets a new vertex point for the vertex whose index is *vertexIndex*. The rounding size is unaffected.

  Parameters

  vertexIndex The index of the vertex you wish to set.

  newVertexPoint The new vertex point.

Throws

ccShapesError::BadGeom
If the resulting general polygon contains any crossing segments.

ccShapesError::BadRadius
If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

ccShapesError::BadCoef
If *vertexIndex* is not a valid vertex index.

ccShapesError::BadGeom
If **isMutable()** = *false*.

Notes

The general polygon is unaffected in case of any throw.

vertexRoundingSize

```
double vertexRoundingSize(c_Int32 vertexIndex) const;

void vertexRoundingSize(
  c_Int32 vertexIndex,
  double newVertexRoundingSize);
```

- `double vertexRoundingSize(c_Int32 vertexIndex) const;`

  Returns the vertex rounding size for the vertex specified by *vertexIndex*.

  Parameters

  vertexIndex The index of the vertex whose rounding size you want returned.

- ```
  void vertexRoundingSize(
    c_Int32 vertexIndex,
    double newVertexRoundingSize);
  ```

  Sets a new vertex rounding size for the vertex specified by *vertexIndex*.

  Parameters

  vertexIndex The index of the vertex whose rounding size you wish to set.

  newVertexRoundingSize
  The new vertex rounding size.

  Throws
  ```

ccShapesError::BadRadius
If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

ccShapesError::BadCoef
If *vertexIndex* is not a valid vertex index.

ccShapesError::BadGeom
If the specified vertex is an end vertex of an open shape and the rounding size is not zero.

ccShapesError::BadGeom
If **isMutable()** = *false*.

Notes

The general polygon is unaffected in case of any throw.

vertexRoundingSizes

```
void vertexRoundingSizes(double newVertexRoundingSize);
```

Sets the rounding sizes for all the vertices in the general polygon.

Parameters

newVertexRoundingSize
The new rounding size.

Throws

ccShapesError::BadRadius
If any of the rounding sizes in the resulting general polygon are invalid (cause an intersection, or are too large).

ccShapesError::BadGeom
If **isMutable()** = *false*.

Notes

The general polygon is unaffected in case of any throw.

roundedVertexArc

```
ccEllipseArc2 roundedVertexArc(c_Int32 vertexIndex) const;
```

Returns the elliptical arc corresponding to the rounded portion of the vertex with the specified index.

Parameters

vertexIndex The vertex index corresponding to the elliptical arc you wish returned.

Throws

ccShapesError::BadCoef
If *vertexIndex* is not a valid vertex index,
or if the specified vertex is not rounded (rounding size = 0).

segmentAngleSpan

```
ccRadian segmentAngleSpan(c_Int32 segmentIndex) const;

void segmentAngleSpan(
  c_Int32 segmentIndex,
  const ccRadian &newSegmentAngleSpan);
```

- `ccRadian segmentAngleSpan(c_Int32 segmentIndex) const;`

Returns the segment angle span for the segment specified by *segmentIndex*.

Parameters

segmentIndex The index of the segment corresponding to the segment angle span you want returned.

- `void segmentAngleSpan(`
  `c_Int32 segmentIndex,`
  `const ccRadian &newSegmentAngleSpan);`

Sets a new segment angle span for the segment specified by *segmentIndex*.

Parameters

segmentIndex The index of the segment corresponding to the segment angle span you wish to set.

newSegmentAngleSpan
The new segment angle span.

Throws

ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

ccShapesError::BadGeom
If this call causes segments to intersect.

Notest

The general polygon is unaffected in case of any throw.

isLineSegment

```
bool isLineSegment(c_Int32 segmentIndex) const;
```

Returns *true* if the segment with the specified index is a line segment (segment angle span equals 0); returns *false* otherwise. This function returns the opposite of **isArcSegment()**.

Parameters

segmentIndex The index of the segment you wish to query.

Throws

ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

isArcSegment

```
bool isArcSegment(c_Int32 segmentIndex) const;
```

Returns *true* if the segment with the specified index is an arc (has a non-zero segment angle span); returns *false* otherwise. This function returns the opposite of **isLineSegment()**.

Parameters

segmentIndex The index of the segment you wish to query.

Throws

ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

ellipseSegment

```
ccEllipseArc2 ellipseSegment(c_Int32 segmentIndex,
  bool ignoreRounding = true) const;
```

Returns the elliptical arc corresponding to the segment with the specified index. If *ignoreRounding* is true, the arc is determined as if there were no rounding at the vertices at each end of the segment. Otherwise, it is determined only by that portion of the segment which is unaffected by the rounding at the vertices at each end of the segment.

Parameters

segmentIndex The index of the segment whose arc you want returned.

ignoreRounding The ignore rounding choice; *true* or *false*.

Throws

ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

ccShapesError::BadGeom
If the specified segment is not an arc (is a line segment).

lineSegment

```
ccLineSeg lineSegment(
  c_Int32 segmentIndex,
  bool ignoreRounding = true) const;
```

Returns the line segment corresponding to the segment with the specified index. If *ignoreRounding* is true, the segment is determined as if there were no rounding at the vertices at each end of the segment. Otherwise, it is determined only by that portion of the segment which is unaffected by the rounding at the vertices at each end of the segment.

Parameters

segmentIndex The index of the segment whose line you want returned.

ignoreRounding The ignore rounding choice; *true* or *false*.

Throws

ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

ccShapesError::BadGeom
If the specified segment is not a line segment (is an arc).

---

map

```
ccGenPoly map(
  const cc2Rigid& c,
  double scale = 1.0) const;

ccGenPoly map(const cc2Xform& c) const;
```

-
```
ccGenPoly map(
  const cc2Rigid& c,
  double scale = 1.0) const;
```

Returns this general polygon scaled by *scale*, and mapped by *c*. The returned general polygon is mutable.

Parameters

c The mapping transform.

scale The scale factor.

Throws

ccMathError::Singular
If *scale* = 0.

- `ccGenPoly map(const cc2Xform& c) const;`

  Returns this general polygon mapped by *c*. The returned general polygon is immutable unless **c.isIdentity()** is *true*.

  Parameters
  c

  Throws
  ccMathError::Singular
  If *c* is singular.

pos `cc2Vect pos() const;`

Returns the position of the general polygon. If there are >0 vertices in this general polygon, the position is defined by the first vertex. Otherwise, it is simply (0,0).

center `cc2Vect center() const;`

Returns the average position of the general polygon (center of mass of all of the vertex points). Returns (0,0) if there are no vertices in this general polygon.

previousVertexIndex

```
c_Int32 previousVertexIndex(c_Int32 segmentIndex) const;
```

Returns the index of the vertex that precedes the segment with index *segmentIndex*.

Parameters
segmentIndex The segment index.

Throws
ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

nextVertexIndex

```
c_Int32 nextVertexIndex(c_Int32 segmentIndex) const;
```

Returns the index of the vertex that follows the segment with index *segmentIndex*.

Parameters
segmentIndex The segment index.

Throws
ccShapesError::BadCoef
If *segmentIndex* is not a valid segment index.

previousSegmentIndex

```
c_Int32 previousSegmentIndex(c_Int32 vertexIndex) const;
```

Returns the index of the segment that precedes the vertex with index *vertexIndex*. Returns -1 if there is no such segment.

Parameters

vertexIndex The vertex index.

Throws

ccShapesError::BadCoef
If *vertexIndex* is not a valid vertex index.

nextSegmentIndex

```
c_Int32 nextSegmentIndex(c_Int32 vertexIndex) const;
```

Returns the index of the segment that follows the vertex with index vertexIndex. Returns -1 if there is no such segment.

Parameters

vertexIndex The vertex index.

Throws

ccShapesError::BadCoef
If *vertexIndex* is not a valid vertex index.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this generalized polygon.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if and only if this generalized polygon is open and has at least one vertex. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if and only if this generalized polygon is closed and has at least one vertex. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For generalized polygons and **cc2Wireframe**s (derived from **ccGenPoly**), this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if this generalized polygon has no vertices, otherwise false. See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

This function returns true for most generalized polygons. It returns false if this generalized polygon contains no line segments or arcs. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

For generalized polygons, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible `virtual bool isReversible() const;`

For generalized polygons, this function always returns true. See **ccShape::reverse()** for more information.

boundingBox `virtual ccRect boundingBox() const;`

Returns the smallest rectangle that encloses this generalized polygon.

Throws

ccShapesError::EmptyShape
isEmpty() returns true.

See **ccShape::boundingBox()** for more information.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const;`

Returns the nearest point on the boundary of this generalized polygon to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p The point.

Throws

ccShapesError::EmptyShape
isEmpty() returns true.

See **ccShape::nearestPoint()** for more information.

startPoint `virtual cc2Vect startPoint() const;`

Returns the starting point of this generalized polygon if it is an open contour.

Throws

ccShapesError::NotOpenContour
This generalized polygon is not an open contour.

See **ccShape::startPoint()** for more information.

endPoint `virtual cc2Vect endPoint() const;`

Returns the ending point of this generalized polygon if it is an open contour.

Throws

ccShapesError::NotOpenContour
This generalized polygon is not an open contour.

See **ccShape::end Point()** for more information.

startAngle `virtual ccRadian startAngle() const;`

Returns the starting tangent direction of this generalized polygon if it is an open contour.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this generalized polygon.

ccShapesError::NoTangent
hasTangent() is false for this generalized polygon.

See **ccShape::startAngle()** for more information.

endAngle `virtual ccRadian endAngle() const;`

Returns the ending tangent direction of this generalized polygon if it is an open contour.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this generalized polygon.

ccShapesError::NoTangent
hasTangent() is false for this generalized polygon.

See **ccShape::endAngle()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this generalized polygon. See **ccShape::reverse()** for more information.

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Returns the net signed angle through which the tangent vector rotates from the start point to the end point.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this generalized polygon.

ccShapesError::NoTangent
hasTangent() is false for this generalized polygon.

See **ccShape::tangentRotation()** for more information.

windingAngle

```
virtual ccRadian windingAngle(const cc2Vect &p) const;
```

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve.

Parameters

p The start point of the vector *p->t* whose angle is measured as the end point *t* traces the curve.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this generalized polygon.

See **ccShape::windingAngle()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true of the given point is within this generalized polygon.

Parameters

p The point.

See **ccShape::within()** for more information.

isRightHanded

```
bool isRightHanded() const;
```

Returns true if and only if this generalized polygon is right-handed.

Throws

ccShapesError::NotRegion
This generalized polygon is not a region.

Notes

The value returned by this function is undefined for self-intersecting polygons.

See **ccShape::isRightHanded()** for more information.

sample

```
void sample(const ccShape::ccSampleParams &params,
 ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores generalized polygons for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

Throws

ccShapesError::SampleOverflow
Supplied spacing and tolerance bounds require more than *maxPoint* samples to be generated. See **ccSampleParams** for details.

mapShape

```
ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this generalized polygon mapped by *X*.

Parameters

X	The transformation object.

Notes

The returned shape may be a **ccContourTree**.

See **ccShape::mapShape()** for more information.

decompose

```
ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccLineSeg**s and **ccEllipseArc2**s. See **ccShape::decompose()** for additional information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

ccGenPoly

```
ccGenPoly(const ccPolygon&);
ccGenPoly(const ccEllipse&);
ccGenPoly(const ccEllipseArc&);
```

Use conversion constructors that use references to supported shapes as arguments instead of these versions.

roundedVertex

```
ccEllipseArc roundedVertex(c_Int32 vertexIndex) const;
```

Use **roundedVertexArc()** instead of this function.

arcSegment

```
ccEllipseArc arcSegment(c_Int32 segmentIndex,
  bool ignoreRounding = true) const;
```

Use **ellipseSegment()** instead of this function.

encloseRect

```
ccRect encloseRect() const;
ccRect encloseRect(const cc2Xform& xform) const;
```

Both overloads of this function are deprecated. Use **boundingBox()** instead.

distToPoint

```
double distToPoint(const cc2Vect & v) const;
```

Use **distanceToPoint()** instead of this function.

ccGenRect

```
#include <ch_cvl/shapes.h>

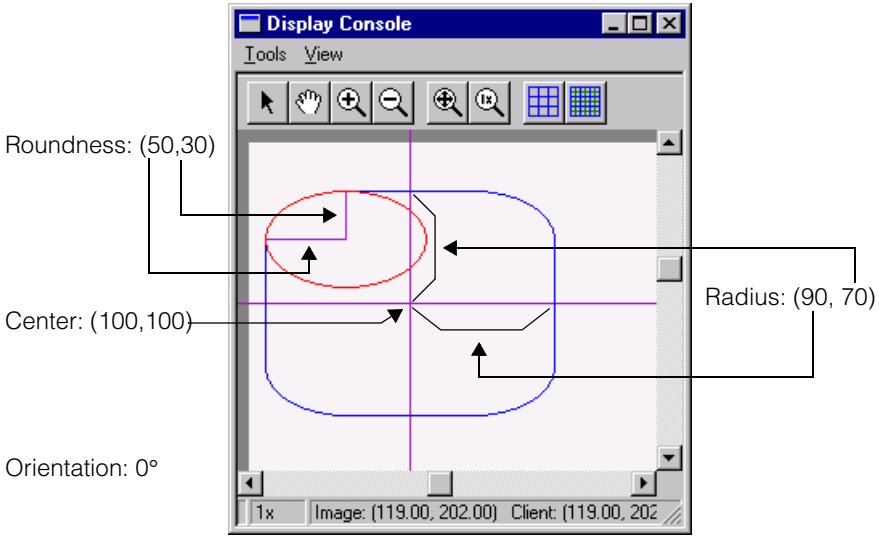
class ccGenRect : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccGenRect** class describes generalized rectangle. Unlike rectangles created by **ccRect**, generalized rectangles can be rotated. and can have rounded edges. To draw a generalized rectangle, you specify a center point, the radii of the rectangle, the roundness of the corners, and an orientation. The following figure illustrates a generalized rectangle.

```
ccGenRect(cc2Vect(100,100),cc2Vect(90,70),ccRadian(0.0),
          cc2Vect(50,30))
```



The **ccGenRect** constructors provide other ways of creating a generalized rectangle.

Note The **ccGenRect** class is immutable except for assignment. As such, it has neither setters nor any non-const members other than constructors, assignment operators, and serialization methods. Once a **ccGenRect** is constructed, the only way to change it is to assign to it or to serialize it (that is, to load to it from an archive).

Constructors/Destructors

ccGenRect

```
ccGenRect();

ccGenRect(const ccGenRect& gr);

ccGenRect(const cc2Vect& cnt, const cc2Matrix& U,
  const cc2Vect& uround);

ccGenRect(const cc2Vect& cnt, const cc2Vect& radii,
  const ccRadian& orient, const cc2Vect& round);

ccGenRect(const ccRect& r);

ccGenRect(const ccEllipse2& e);
```

- `ccGenRect();`

  The default constructor creates a degenerate rectangle.

  Notes

  The default constructor is provided to support arrays and STL containers. Do not try to examine or use a default-constructed **ccGenRect**.

- `ccGenRect(const ccGenRect& gr);`

  Copy constructor. Creates a copy of an existing generalized rectangle.

  Parameters

gr	The existing generalized rectangle, which must not be degenerate.

- ```
  ccGenRect(const cc2Vect& cnt, const cc2Matrix& U,
    const cc2Vect& uround);
  ```

  Creates a generalized rectangle by applying the matrix *U* and the unit vector *uround* (for corner roundness) to an ideal 1 x 1 rectangle.

  Let R be the constructed generalized rectangle (ignoring rounding for the moment), and let R' be a translated version of R centered at the origin. Finally, let C be a canonical 2 x 2 rectangle over the range [-1, +1] x [-1, +1]. The matrix *U* specifies the map from R' to C. Hence, the constructed generalized rectangle (without rounding) if generated by mapping the canonical rectangle *C* by **U.inverse()** and then translating by *cnt*.

  The normalized roundness vector specifies the degree of corner rounding in the x- and y-directions of the canonical rectangle *C* (not of *R*). Therefore, the components of *uround* are meaningful over the range 0 through 1. A roundness value of 0.0 specifies no corner rounding. A roundness value of 1.0 specifies complete corner rounding. For example, if **uround.x()** == 1.0, the x sides of the canonical rectangle will consist entirely of rounded corners, with no straight segment between them. The constructed generalized rectangle (with rounding) is generated by mapping the rounded canonical rectangle *C* by **U.inverse()** and then translating by *cnt*.

  Since *U* may include scaling, rotation, skew, and aspect, this is the most general constructor for generalized rectangles. It is recommended, however, that generalized rectangles with skew be avoided. Although geometric operations on such generalized rectangles will be correct, they may not display correctly.

  Parameters

cnt	The center of the rectangle.
U	A matrix that specifies the rotation and scale of the rectangle.
uround	The roundness of the rectangle's corners.

- ```
  ccGenRect(const cc2Vect& cnt, const cc2Vect& radii,
    const ccRadian& orient, const cc2Vect& round);
  ```

  Creates a generalized rectangle given a center, two radii, and orientation, and roundness as shown in the illustration at the beginning of this reference page.

  The roundness vector specifies the degree of corner rounding in the x- and y-directions of the generalized rectangle. The value **round.x()** is meaningful over the range 0 through **radii.x()**; other values are clipped to this range internally. The analogous statement holds for **round.y()** and **radii.y()**. A roundness value of 0.0 specifies no corner rounding. A roundness value equal to the corresponding radius value specifies complete corner rounding. For example, if **round.x()** == **radii.x()**, the x-sides of the generalized rectangle will consist entirely of rounded corners, with no straight segment between them.

Parameters

cnt The center of the rectangle.

radii The radii of the rectangle (half-height and half-width).

orient The orientation of the rectangle measured from the x-axis.

round The roundness of the rectangle's corners.

Throws

ccShapesError::BadRadius
If either component of radii is not positive.

- `ccGenRect(const ccRect& r);`

Construct a generalized rectangle that has the same geometry as the rectangle *r*. The rectangle *r* must not be degenerate.

Parameters

r The rectangle to copy.

- `ccGenRect(const ccEllipse2& e);`

Construct a generalized rectangle that has the same geometry as the ellipse *e*. The ellipse *e* must not be degenerate.

Parameters

e The ellipse to copy.

Throws

ccShapesError::BadRadius
One of the ellipse radii is less than zero.

Operators

operator== `bool operator==(const ccGenRect& that) const;`

Returns true if this generalized rectangle is equal to that, false otherwise.

Parameters

that The other generalized rectangle.

operator!= `bool operator!=(const ccGenRect& that) const;`

Returns true if this generalized rectangle is not equal to that, false otherwise.

Parameters

that The other generalized rectangle.

Public Member Functions

center

```
const ccPoint& center() const;
```

Returns the center of this generalized rectangle.

U

```
const cc2Matrix& U() const;
```

Returns the matrix used to transform the ideal 1x1 rectangle into a generalized rectangle.

radii

```
const cc2Vect& radii() const;
```

Returns the radii (half-height and half-width) of the generalized rectangle.

orient

```
ccRadian orient() const;
```

Returns the orientation of the generalized rectangle measured from the x-axis.

skew

```
ccRadian skew() const;
```

Returns the amount by which the generalized rectangle is skewed.

round

```
cc2Vect round() const;
```

Returns the roundness of the corners of the generalized rectangle.

uround

```
const cc2Vect& uround() const;
```

Returns the roundness of the generalized rectangle as a unit vector.

map

```
ccGenRect map(const cc2Xform& c) const;
```

Returns a generalized rectangle that is the result of mapping this generalized rectangle with the transformation object *c*.

Parameters

c The transformation object.

Notes

The transform *c* must be nonsingular. If it is not, use **mapshape()** instead.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this generalized rectangle.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For generalized rectangles, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For generalized rectangles, this function always returns true, even for generalized rectangles that are degenerate. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For generalized rectangles, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For generalized rectangles, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

This function returns true if the perimeter of this generalized rectangle is positive. It returns false if the perimeter is zero. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For generalized rectangles, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For generalized rectangles, this function always returns false. See **ccShape::reverse()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

Returns true if this generalized rectangle is right-handed. See **ccShape::isRightHanded()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this generalized rectangle. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this generalized rectangle to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores generalized rectangles for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this generalized rectangle mapped by *X*.

Parameters

X	The transformation object.

Notes

If the transform *X* is singular, this function returns a closed **ccContourTree**. Otherwise this function returns a **ccGenRect**.

See **ccShape::mapShape()** for more information.

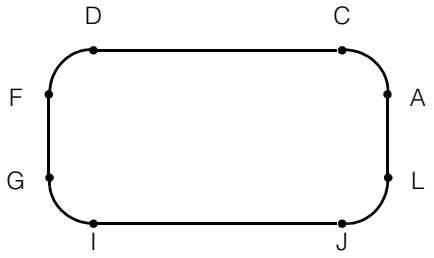
decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a closed **ccContourTree** consisting of connected **ccLineSeg**s and **ccEllipseArc2**s.

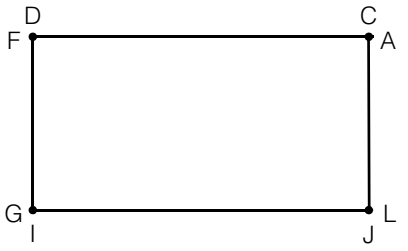
If the corners are rounded, the contour tree will have eight children in the following order:

1. Arc AC
2. Segment CD
3. Arc DF
4. Segment FG
5. Arc GI
6. Segment IJ
7. Arc JL
8. Segment LA



If the corners are not rounded, the contour tree will have four segment children in the following order:

1. Segment CD
2. Segment FG
3. Segment IJ
4. Segment LA



See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true of the given point is within this generalized polygon.

Parameters

p	The point.

See **ccShape::within()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

ccGenRect

```
ccGenRect(const ccEllipse& e);
```

Use the conversion constructor that uses a **ccEllipse2** reference instead of this version.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

ccGenRectData

```
struct ccGenRectData {
  ccPoint S;
  ccLineSeg CD, FG, IJ, LA;
  ccEllipseArc AC, DF, GI, JL;
  ccGenRectData ();
  ccGenRectData (const ccGenRect&);
  ccGenRectData (const ccGenRectData& grd,
     const cc2Xform &c);
};
```

Both the **ccGenRect::ccGenRectData** structure and the following function, which retrieves it, are deprecated.

genRectData

```
const ccGenRectData& genRectData() const;
```

ccGigEVisionCamera

```
#include <ch_cvl/vpgige.h>

class ccGigEVisionCamera : public ccUnknownFG
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes an individual GigE Vision-compliant camera connected to a computer system. There is one instance of this class for each connected camera. You use the static **get()** function to obtain a specific instance.

```
ccGigEVisionCamera& GECam = ccGigEVisionCamera::get(0);
```

Calling the **name()** function (a member of **ccBoard**, from which this class is derived) returns the name of the particular camera,

Many commonly used GigE vision camera features are supported directly through CVL image acquisition properties. For features that are not supported directly by CVL you can use the feature read/write functions such as **writeIntegerValue()** to access them directly.

In particular you should not use the feature read/write functions to modify the following GigE Vision features. If you do, the CVL acquisition system may behave erratically.

GigE Vision features used by CVL

AcquisitionMode	AcquisitionStart	AcquisitionStop
PixelFormat	TriggerMode	
ExposureTimeAbs	ExposureTimeRaw	
BlackLevelRaw	GainRaw	
OffsetX	OffsetY	
Width	Height	

Table 1.

Constructors/Destructors

A single instance of this class is created automatically for each GigE Vision camera connected to your system.

Public Member Functions

count

```
static c_Int32 count();
```

Returns the number of GigE Vision cameras found on the network.

Notes

If the cameras have not yet finished negotiating their connection with the network, this function may take up to several minutes to return.

get

```
static ccGigEVisionCamera& get(c_Int32 i = 0);
```

Returns a **ccGigEVisionCamera** object that represents the specified camera. The association between specific GigE Vision cameras and their index value is defined by the camera's IP address. You can also use the **name()** and **serialNumber()** functions to identify the camera returned by this function.

Parameters

i The index of the camera to get.

Throws

ccBoard::BadParams

i is less than zero or greater than or equal to **count()**.

Notes

For any given index value, the object returned by this function and the object returned by calling **ccBoard::get()** with the same index may not refer to the same device, since **ccBoard::get()** returns references to *all* devices, such as Cognex frame grabbers, not just GigE Vision cameras.

serialNumber

```
virtual ccCvlString serialNumber() const;
```

Returns the serial number of the GigE Vision camera. The format of the returned string is specific to the camera type.

resetTimeStamp

```
void resetTimeStamp();
```

Resets the timestamp counter in the camera to 0.

readTimeStamp

```
c_UInt64 readTimeStamp();
```

Returns the instantaneous timestamp counter value. The timestamp is an arbitrary counter. It was no relationship to clock time.

timeStampFrequency

```
c_UInt64 timeStampFrequency();
```

Returns the number timestamp counts per second.

readIntegerValue

```
c_Int32 readIntegerValue(ccCvlString nodeName);
```

Returns the integer value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name.

writeIntegerValue

```
void writeIntegerValue(ccCvlString nodeName,
   c_Int32 value);
```

Writes the integer value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

value The value to write.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name or *value* was not a valid value for this node.

readDoubleValue

```
double readDoubleValue(ccCvlString nodeName);
```

Returns the double value from associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name.

writeDoubleValue

```
void writeDoubleValue(ccCvlString nodeName, double value);
```

Writes the double value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

value The value to write.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name or *value* was not a valid value for this node.

readEnumValue

```
ccCvlString readEnumValue(ccCvlString nodeName);
```

Returns the enum value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name.

writeEnumValue

```
void writeEnumValue (ccCvlString nodeName,
                     ccCvlString value);
```

Writes the enum value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

value The value to write.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name or *value* was not a valid value for this node.

readValue

```
ccCvlString readValue(ccCvlString nodeName);
```

Returns the value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name.

writeValue

```
void writeValue (ccCvlString nodeName,
                 ccCvlString value);
```

Writes the value associated with the specified node name in the camera's definition file.

Parameters

nodeName The name of the node.

value The value to write. For nodes that describe Boolean values, the strings "1" and "0" are interpreted as True and False.

Notes

Use this function to access nodes for features that are not implemented directly in CVL. See Table 1 on page 1549 for a list of these features.

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name or *value* was not a valid value for this node.

executeCommand

```
void executeCommand(ccCvlString nodeName);
```

Executes the command specified by the *nodeName*.

Parameters

nodeName The name of the node.

Example

To save the current settings as "UserSet1", you would use the following code. (*fg* is a **ccGigEVisionCamera** object.)

```
fg.writeEnumValue(cmT("UserSetSelector"), cmT("UserSet1"));
fg.executeCommand(cmT("UserSetSave"));
```

Throws

ccGigEVisionCamera::FeatureError

nodeName was not a valid node name.

getCurrentIPAddress

```
void getCurrentIPAddress(c_Int32* addr,
                         c_Int32* subnetMask,
                         bool* isPersistent);
```

Gets the current IP address of the camera.

Parameters

addr	The IP address of the camera.
subnetMask	The subnet mask of the camera.
isPersistent	True if the address is persistent; False otherwise.

getHostIPAddress

```
void getHostIPAddress(c_Int32* addr, c_Int32* subnetMask);
```

Returns the IP address of the NIC the camera is attached to.

readTimeout

```
void readTimeout(const double seconds);

double readTimeout() const;
```

- `void readTimeout(const double seconds);`

  Sets the amount of time to wait in seconds for an Acknowledgement from the camera after a read operation is started. This includes the network communication time as well as the amount of time required by the camera to perform the requested operation.

  Throws

  ccBoard::BadParams
  The read timeout value is less than 1.0 second (1000 milliseconds).

- `double readTimeout() const;`

  Gets the amount of time to wait in seconds for an Acknowledgement from the camera after a read operation is started. This includes the network communication time as well as the amount of time required by the camera to perform the requested operation.

  Notes

  The default timeout value for reads is 1.0 second as recommended by the GigE Vision standard. This is also the minimum timeout value. There is no maximum value.

  Some cameras may require a timeout period longer than the default for some operations.

  If a camera read timeout occurs, a NetworkError exception will be thrown with the description "Receive Timeout".

writeTimeout

```
void writeTimeout(const double seconds);
double writeTimeout() const;
```

- `void writeTimeout(const double seconds);`

  Sets the amount of time to wait in seconds for an Acknowledgement from the camera after a write operation is started. This includes the network communication time as well as the amount of time required by the camera to perform the requested operation.

  Throws

  ccBoard::BadParams

  The write timeout value is less than 1.0 second (1000 milliseconds).

- `double writeTimeout() const;`

  Gets the amount of time to wait in seconds for an Acknowledgement from the camera after a write operation is started. This includes the network communication time as well as the amount of time required by the camera to perform the requested operation.

  Notes

  The default timeout value for writes is 1.0 second as recommended by the GigE Vision standard. This is also the minimum timeout value. There is no maximum value.

  Some cameras may require a timeout period longer than the default for some operations.

  If a camera write timeout occurs, a NetworkError exception will be thrown with the description "Receive Timeout".

Global Exceptions

These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareInUse

The current process tried to access a camera that is already owned by another running process. To avoid this error, a process that touches the camera must exit before another process can access the same hardware.

ccBoard::HardwareNotResponding

A camera communication error occurred.

ccGigEVisionCamers::NetworkError
: A camera on the network failed to respond. A network cable may be disconnected.

ccGigEVisionTransportProp

```
#include <ch_cvl/prop.h>

class ccGigEVisionTransportProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccGigEVisionTransportProp** class controls the transport-related properties for GigE Vision cameras.

Public Member Functions

transportTimeout

```
void transportTimeout(double val);

double transportTimeout() const;

static const double defaultTransportTimeout;
```

- `void transportTimeout(double val);`

  Sets the maximum timeout, in seconds, for the camera to transmit an image.

  Parameters

  val The timeout value to set.

  Throws

  ccGigEVisionTransportProp::BadParams
  Transport timeout < 0

  Notes

  The time interval begins when the camera sends the first data packet. This means that transportTimeout does not need to account for exposure time or for time spent waiting for a trigger signal.

  The most common use case is to adjust the transport timeout to a larger value to allow successful acquisition from cameras with a long readout time, as may occur with a linescan camera.

- `double transportTimeout() const;`

  Gets the maximum timeout.

- `static const double defaultTransportTimeout;`

  Gets the default maximum timeout, which is 2.0 seconds.

packetSize

```
void packetSize(c_Int32 val);
c_Int32 packetSize() const;
c_Int32 packetSizeMax() const;
```

- `void packetSize(c_Int32 val);`

  Sets the size of image packets transmitted by the camera.

  Parameters

  val Image packet size to set.

  Throws

  ccGigEVisionTransportProp::BadParams
  The value is less than 1 or greater than the maximum.

  Notes

  The default is the value that was automatically determined, and will vary based on the camera used and the network configuration. In most cases, the packetSize value should not be changed from the default.

  Ideally, the automatically determined value will be 8000 or larger. A smaller value may indicate that jumbo frames are disabled on the network adapter, or that a switch in the network does not support jumbo frames.

  The packetSize function controls the GevSCPSPacketSize feature of the camera.

- `c_Int32 packetSize() const;`

  Gets the size of image packets transmitted by the camera.

- `c_Int32 packetSizeMax() const;`

  Gets the maximum packet size.

latencyLevel

```
void latencyLevel(c_Int32 val);

c_Int32 latencyLevel() const;

static const c_Int32 defaultLatencyLevel;
```

- `void latencyLevel(c_Int32 val);`

  Sets the latency level of the driver.

  Parameters

  val Latency level to set.

  Throws

  ccGigEVisionTransportProp::BadParams
  The value is less than 0 or greater than 3.

  Notes

  The valid range is 0 (lowest latency) to 3 (lowest CPU).

  Smaller values reduce acquisition latency and may improve reliability at the expense of higher CPU usage.

- `c_Int32 latencyLevel() const;`

  Gets the latency level of the driver.

- `static const c_Int32 defaultLatencyLevel;`

  Gets the default latency level of the driver, which is 3.

ccGigEVisionTransportProp

ccGMorph3x3Element

```
#include <ch_cvl/gmorph.h>

class ccGMorph3x3Element;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single 3x3 pixel grey-scale morphology structuring element.

Constructors/Destructors

ccGMorph3x3Element

```
ccGMorph3x3Element(ccGMorphDefs::ElementType
  type = ccGMorphDefs::eSquare3x3);
```

Constructs a 3x3 structuring element of the specified type. All offsets in the element are set to 0.

Parameters

type The element type to create. *type* must be one of the following values:

ccGMorphDefs::eSquare3x3
ccGMorphDefs::eDiamond3x3
ccGMorphDefs::eLine1x3_0
ccGMorphDefs::eLine1x3_45
ccGMorphDefs::eLine1x3_90
ccGMorphDefs::eLine1x3_135
ccGMorphDefs::eArbitrary

Operators

operator==

```
bool operator== (const ccGMorph3x3Element& that) const;
```

Tests this **ccGMorph3x3Element** for equality with the supplied one. The function returns true if the two **ccGMorph3x3Element**s are of the same type and have the same offsets, mask, and origins.

Parameters

that The **ccGMorph3x3Element** to compare to this one

Public Member Functions

elementType

```
const ccGMorphDefs::ElementType elementType() const;
void elementType(const ccGMorphDefs::ElementType& type);
```

- `const ccGMorphDefs::ElementType elementType() const;`

  Returns the type of this **ccGMorph3x3Element**. The returned value is one of the following values:

  ccGMorphDefs::eSquare3x3
  ccGMorphDefs::eDiamond3x3
  ccGMorphDefs::eLine1x3_0
  ccGMorphDefs::eLine1x3_45
  ccGMorphDefs::eLine1x3_90
  ccGMorphDefs::eLine1x3_135
  ccGMorphDefs::eArbitrary

- `void elementType(const ccGMorphDefs::ElementType& type);`

  Sets the type of this **ccGMorph3x3Element**.

  Parameters

  type The element type to set. *type* must be one of the following values:

  ccGMorphDefs::eSquare3x3
  ccGMorphDefs::eDiamond3x3
  ccGMorphDefs::eLine1x3_0
  ccGMorphDefs::eLine1x3_45
  ccGMorphDefs::eLine1x3_90
  ccGMorphDefs::eLine1x3_135
  ccGMorphDefs::eArbitrary

offsets

```
cmStd vector<c_Int8> offsets() const;

void offsets(cmStd vector<c_Int8>& values);
```

- `cmStd vector<c_Int8> offsets() const;`

  Returns a nine-element vector of **c_Int8** containing the offsets for each pixel of this **ccGMorph3x3Element**. The offsets are returned in the following order: NW, N, NE, W,C,E, SW, S, and SE. Each offset is a signed value in the range -128 to 127. Returns a vector of zeros for all of the predefined element types.

- `void offsets(cmStd vector<c_Int8>& values);`

  Sets the offsets for each pixel of this **ccGMorph3x3Element**. The offsets must be supplied in the following order: NW, N, NE, W,C,E, SW, S, and SE. Each offset must be a signed value in the range -128 to 127.

  Parameters

  values The offsets to set.

  Throws

  ccGmorphDefs::EmptyList
  No 3x3 elements are present.

  Notes

  The setter should be used only if the *elementType* is *eArbitrary*.
  If this setter is used, the offsets are interpreted as additive offsets.

dontCareMask

```
c_UInt32 dontCareMask() const;

void dontCareMask(const c_UInt32 maskValue);
```

- `c_UInt32 dontCareMask() const;`

  Returns the don't care mask for this **ccGMorph3x3Element**. The returned value is constructed by ORing together each of the following values that corresponds to a don't care pixel:

  ccGMorphDefs::ePosNE
  ccGMorphDefs::ePosN
  ccGMorphDefs::ePosNW
  ccGMorphDefs::ePosW
  ccGMorphDefs::ePosSW
  ccGMorphDefs::ePosS

ccGMorphDefs::ePosSE
ccGMorphDefs::ePosE
ccGMorphDefs::ePosCntr

If no don't care mask has been set, this function returns 0 (all pixels are care pixels).

- `void dontCareMask(const c_UInt32 maskValue);`

Sets the don't care mask for this **ccGMorph3x3Element**. The mask is specified by ORing together each of the values defined in **ccGMorphDefs::Pos3x3** that corresponds to a don't care pixel.

Parameters

maskValue The mask to set. *maskValue* must be either 0 or a value formed by ORing together any of the following values:

ccGMorphDefs::ePosNE
ccGMorphDefs::ePosN
ccGMorphDefs::ePosNW
ccGMorphDefs::ePosW
ccGMorphDefs::ePosSW
ccGMorphDefs::ePosS
ccGMorphDefs::ePosSE
ccGMorphDefs::ePosE
ccGMorphDefs::ePosCntr

Notes

The setter should be used only if the *elementType* is *eArbitrary*.
If this setter is used, the offsets are interpreted as masking offsets.

The offsets can be modified and interpreted as listed below:

Element Type	Setter to Modify Offsets	Values Interpreted as
eArbitrary	**offsets(***values***)**	additive
eArbitrary	**dontCareMask(***mask***)**	mask
all predefined	do not modify offsets	mask (always)

origin

```
ccGMorphDefs::Pos3x3 origin() const;
void origin(const ccGMorphDefs::Pos3x3 pos);
```

- `ccGMorphDefs::Pos3x3 origin() const;`

  Returns the origin of this **ccGMorph3x3Element**. The returned value is one of the following:

  ccGMorphDefs::ePosNE
  ccGMorphDefs::ePosN
  ccGMorphDefs::ePosNW
  ccGMorphDefs::ePosW
  ccGMorphDefs::ePosSW
  ccGMorphDefs::ePosS
  ccGMorphDefs::ePosSE
  ccGMorphDefs::ePosE
  ccGMorphDefs::ePosCntr

- `void origin(const ccGMorphDefs::Pos3x3 pos);`

  Sets the origin of this **ccGMorph3x3Element**.

  Parameters

  pos The origin to set. *pos* must be one of the following values:

  ccGMorphDefs::ePosNE
  ccGMorphDefs::ePosN
  ccGMorphDefs::ePosNW
  ccGMorphDefs::ePosW
  ccGMorphDefs::ePosSW
  ccGMorphDefs::ePosS
  ccGMorphDefs::ePosSE
  ccGMorphDefs::ePosE
  ccGMorphDefs::ePosCntr

  The default origin is *ccGMorphDefs::ePosCntr*.

  Throws

  ccGMorphDefs::BadElement
  The origin is outside the structuring element.

ccGMorph3x3Element

ccGMorphDefs

```
#include <ch_cvl/gmorph.h>

class ccGMorphDefs;
```

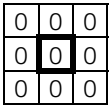
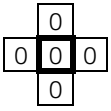
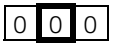
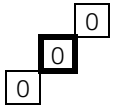
A name space that holds enumerations and constants used with the Grey-scale Morphology tool.

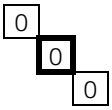
Enumerations

ElementType

```
enum ElementType
```

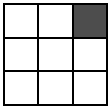
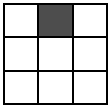
This enumeration defines the pre-defined 3x3 structuring element types supported by the Grey-scale Morphology tool.

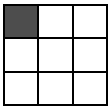
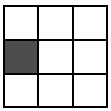
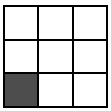
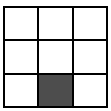
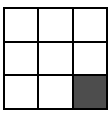
Value	Meaning
eSquare3x3	 3x3 square
eDiamond3x3	 3x3 diamond
eLine1x3_0	 1x3 horizontal line (0° from horizontal)
eLine1x3_45	 1x3 diagonal line (45° from horizontal)

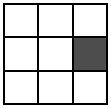
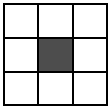
Value	Meaning
eLine1x3_90	 1x3 vertical line (90° from horizontal)
eLine1x3_135	 1x3 diagonal line (135° from horizontal)
eArbitrary	The element is undefined.
kDefaultElementType	The default element type, as defined in *gmorph.h*

Pos3x3 `enum Pos3x3`

This enumeration defines the pixel locations within a 3x3 structuring element.

Value	Meaning
ePosNE	
ePosN	

Value	Meaning
ePosNW	
ePosW	
ePosSW	
ePosS	
ePosSE	

Value	Meaning
ePosE	
ePosCntr	

ccGMorphElement

```
#include <ch_cvl/gmorph.h>

class ccGMorphElement : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains a grey-scale morphology structuring element of any size. It includes members that let you compose a structuring element and members that let you apply the structuring element to an input image.

Constructors/Destructors

ccGMorphElement

```
ccGMorphElement();

ccGMorphElement(
  const cmStd vector<ccGMorph3x3Element>& elements);

virtual ~ccGMorphElement();

ccGMorphElement(const ccGMorphElement& rhs);
```

- `ccGMorphElement();`

  Constructs a **ccGMorphElement** that cannot be applied to an image.

- ```
  ccGMorphElement(
    cmStd vector<ccGMorph3x3Element>& elements);
  ```

  Composes a structuring element using the supplied list of 3x3 structuring elements. All but the first 3x3 element are reflected, then applied to the first 3x3 element using dilation.

  Parameters

  elements A vector of **ccGMorph3x3Element**.

  Throws
  ```

ccGMorphDefs::BadElement
An invalid 3x3 element is present in *elements*.

Operators

operator==

```
bool operator== (const ccGMorphElement& that) const;
```

Tests this **ccGMorphElement** for equality with the supplied one. The function returns true if the **ccGMorphElement**s have the same height, width, and offsets.

Parameters
that The **ccGMorphElement** to compare to this one

Public Member Functions

elementArray

```
cmStd vector<ccGMorph3x3Element> elementArray() const;

void elementArray(
  const cmStd vector<ccGMorph3x3Element>& elements);
```

- ```
  cmStd vector<ccGMorph3x3Element> elementArray() const;
  ```

  Returns a vector of **ccGMorph3x3Element** These 3x3 structuring elements were used to compose this **ccGMorphElement**.

  Throws
  ccGMorphDefs::EmptyList
  This **ccGMorphElement** contains no 3x3 structuring elements.

- ```
  void elementArray(
    const cmStd vector<ccGMorph3x3Element>& elements);
  ```

  Sets the 3x3 structuring elements used to compose this **ccGMorphElement**.

  Parameters
  elements A vector of **ccGMorph3x3Element**.

  Throws
  ccGMorphDefs::BadElement
  An invalid 3x3 element is present in *elements*.

num3x3Elements

```
c_Int32 num3x3Elements() const;
```

Returns the number of 3x3 elements which were used to compose this **ccGMorphElement**.

renderElement

```
ccPelBuffer_const<c_UInt8> renderElement() const;
```

Return a **ccPelBuffer_const<c_UInt8>** that contains a graphical representation of this **ccGMorphElement**.

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

origin

```
ccIPair origin() const;
```

```
void origin(const ccIPair& pos);
```

- `ccIPair origin() const;`

  Returns the origin of this **ccGMorphElement** in unreflected integer coordinates. The upper-left corner of the element is at (0,0).

- `void origin(const ccIPair& pos);`

  Sets the origin of this **ccGMorphElement** in unreflected integer coordinates. The upper-left corner of the element is at (0,0).

  Parameters

  pos The origin to set.

  Throws

  ccGMorphDefs::EmptyList
  This **ccGMorphElement** contains no 3x3 structuring elements.

  ccGMorphDefs::BadElement
  The origin is outside the structuring element.

width

```
c_Int32 width() const;
```

Returns the width of this **ccGMorphElement**.

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

height

```
c_Int32 height() const;
```

Returns the height of this **ccGMorphElement**.

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

erode

```
void erode(const ccPelBuffer_const<c_UInt8>& srcImg,
 ccPelBuffer<c_UInt8>& dstImg) const;
```

Perform an erosion by applying this **ccGMorphElement** to the supplied image.

Parameters

srcImg The input image

dstImg The output image

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

ccGMorphDefs::BadImage
srcImg is not large enough to apply the structuring element.

dilate

```
void dilate(const ccPelBuffer_const<c_UInt8>& srcImg,
 ccPelBuffer<c_UInt8>& dstImg) const;
```

Perform a dilation by applying this **ccGMorphElement** to the supplied image.

Parameters

srcImg The input image

dstImg The output image

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

ccGMorphDefs::BadImage
srcImg is not large enough to apply the structuring element.

open

```
void open(const ccPelBuffer_const<c_UInt8>& srcImg,
 ccPelBuffer<c_UInt8>& dstImg) const;
```

Perform an opening by applying this **ccGMorphElement** to the supplied image.

Parameters

srcImg The input image

dstImg The output image

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

ccGMorphDefs::BadImage
srcImg is not large enough to apply the structuring element.

close

```
void close(const ccPelBuffer_const<c_UInt8>& srcImg,
  ccPelBuffer<c_UInt8>& dstImg) const;
```

Perform a closing by applying this **ccGMorphElement** to the supplied image.

Parameters

srcImg The input image

dstImg The output image

Throws

ccGMorphDefs::EmptyList
This **ccGMorphElement** contains no 3x3 structuring elements.

ccGMorphDefs::BadImage
srcImg is not large enough to apply the structuring element.

ccGMorphElement

ccGraphic

```
#include <ch_cvl/glist.h>

class ccGraphic :  public virtual ccPersistent,
                   public virtual ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

ccGraphic is the base class for graphic items. It defines functions common to the derived classes **ccGraphicBuiltIn**, **ccGraphicEllipseAnnulusSection,** **ccGraphicText**, **ccGraphicPointIcon,** and **ccGraphicCross**.

ccGraphic-derived classes are wrapper classes for graphic shapes defined in *shapes.h* and are stored in a **ccGraphicList** container class to provide an iterable list of graphics.

Constructors/Destructors

ccGraphic

```
ccGraphic();

ccGraphic(const ccColor& clr);

ccGraphic(const ccGraphicProps& props);
```

- `ccGraphic();`

  Creates an empty graphic object.

- `ccGraphic(const ccColor& clr);`

  Creates an empty graphic object of the specified color.

  Parameters

  clr The color to use.

- `ccGraphic(const ccGraphicProps& props);`

  Creates an empty graphic object using the specified graphic properties.

Parameters

props The graphic properties object specifying the color and other drawing properties of the graphic.

Public Member Functions

props

```
const ccGraphicProps& props() const;
ccGraphicProps& props();
void props(const ccGraphicProps& p);
```

- `const ccGraphicProps& props() const;`

  Retrieves the drawing properties of the graphic object as a const object reference.

- `ccGraphicPropsProps& props();`

  Retrieves the drawing properties of the graphic object as a non-const object reference.

- `void props(const ccGraphicProps& p);`

  Sets the drawing properties of the graphic object.

  Parameters

  p The drawing properties.

map

```
virtual ccGraphicPtrh map(const cc2Xform& c) const = 0;
```

Returns a copy of this graphic item shape, mapped by the transformation object *c*.

Parameters

c The transformation object.

clone

```
virtual ccGraphicPtrh clone() const = 0;
```

Duplicates this object and returns a pointer to the duplicate.

encloseRect

```
virtual ccRect encloseRect() const = 0;
```

Returns the enclosing rectangle for the shape of this graphic item.

distToPoint

```
virtual double distToPoint(const cc2Vect& v) const = 0;
```

Returns the distance from the given point to this graphic line.

Parameters

v The point to measure from, as a **cc2Vect** x,y coordinate pair.

Deprecated Members

color

```
const ccColor& color() const;
void color(const ccColor& c);
```

Deprecated function, replaced with **props()**.

ccGraphicBuiltin

```
#include <ch_cvl/glist.h>

template <class T>
class ccGraphicBuiltin : public ccGraphic;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

ccGraphicBuiltin is a template base class for built in **ccGraphic** items that defines functions common to derived classes **ccGraphicSimple** and **ccGraphicWithFill**. All **ccGraphicBuiltin**-derived classes provide wrapper classes for shapes defined in *shapes.h*.

T Template parameter which specifies the type of the graphic item. Graphic item types are specified by a **typedef** in the derived classes **ccGraphicSimple** and **ccGraphicWithFill**.

Constructors/Destructors

ccGraphicBuiltin

```
ccGraphicBuiltin();

ccGraphicBuiltin(const T& item, const ccColor& clr);

ccGraphicBuiltin(const T& item,
  const ccGraphicProps& props);
```

- `ccGraphicBuiltin();`

  Creates a built in graphic.

- `ccGraphicBuiltin(const T& item, const ccColor& clr);`

  Creates a built in graphic of specified type and color.

  Parameters

  item The graphic item.

  clr The color of the graphic.

- `ccGraphicBuiltin(const T& item, const ccGraphicProps& props);`

  Creates a built in graphic of the specified type using the specified drawing properties.

  Parameters

item	The graphic item.
props	The drawing properties.

Public Member Functions

item

```
const T& item() const;
T& item();
```

- `const T& item() const;`

  Returns the underlying shape of this graphic item.

- `T& item();`

  Returns a non-**const** reference to the underlying shape of this graphic item.

Deprecated Members

encloseRect

```
ccRect encloseRect() const;
```

Returns the enclosing rectangle of the underlying shape of this graphic item.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Retrieves the distance from the built in graphic to the specified point.

Parameters

v	Two-dimensional vector specifying the coordinates of the point.

ccGraphicCross

```
#include <ch_cvl/glist.h>

class ccGraphicCross : public ccGraphic;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

ccGraphicCross is a graphic item wrapper class for the **ccCross** class defined in *shapes.h*.

Constructors/Destructors

ccGraphicCross

```
ccGraphicCross();

ccGraphicCross(const ccCross& cross, const ccColor& clr);

ccGraphicCross(const ccCross& cross,
  const ccGraphicProps& props);
```

- `ccGraphicCross();`

  Creates a cross graphic.

- `ccGraphicCross(const ccCross& cross, const ccColor& clr);`

  Creates a cross graphic of the specified color

  Parameters

cross	The cross.
clr	The color.

- ```
  ccGraphicCross(const ccCross& cross,
    const ccGraphicProps& props);
  ```

  Creates a cross graphic using the specified drawing properties.
  ```

Parameters

cross	The cross.
props	The drawing properties.

Public Member Functions

item

```
const ccCross& item() const;
ccCross& item();
```

- `const ccCross& item() const;`

  Returns this graphic item.

- `ccCross& item();`

  Returns a non-**const** reference to this graphic item.

map

```
ccGraphicPtrh map(const cc2Xform& c) const;
```

Returns a copy of this graphic item, mapped by the transformation object *c*.

Parameters

c	The transformation.

clone

```
ccGraphicPtrh clone() const;
```

Duplicates this object and returns a pointer to the duplicate.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Retrieves the distance from the point icon to the specified point.

Parameters

v	Two-dimensional vector specifying the coordinates of the point.

ccGraphicEllipseAnnulusSection

```
#include <ch_cvl/glist.h>

class ccGraphicEllipseAnnulusSection : public ccGraphic;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

ccGraphicEllipseAnnulusSection is a graphic item wrapper class for the **ccEllipseAnnulusSection** class defined in *shapes.h*.

Constructors/Destructors

ccGraphicEllipseAnnulusSection

```
ccGraphicEllipseAnnulusSection();

ccGraphicEllipseAnnulusSection(
  const ccEllipseAnnulusSection& item,
  const ccColor& clr,
  bool arrow = false, bool forward = false);

ccGraphicEllipseAnnulusSection(
  const ccEllipseAnnulusSection& item,
  const ccGraphicProps& props);
```

- ```
  ccGraphicEllipseAnnulusSection();
  ```

  Creates an ellipse annulus section graphic.

- ```
  ccGraphicEllipseAnnulusSection(
    const ccEllipseAnnulusSection& item,
    const ccColor& clr,
    bool arrow = false, bool forward = false);
  ```

  Creates an ellipse annulus section graphic with the specified color, arrow heads, and arrow direction.

  Parameters

  item The ellipse annulus section.

clr	The color.
arrow	Flag to display arrows heads. True displays arrows heads. False hides arrow heads.
forward	Flag for arrow direction. True displays arrow forward. False displays arrow in reverse.

- ```
  ccGraphicEllipseAnnulusSection(
    const ccEllipseAnnulusSection& item,
    const ccGraphicProps& props);
  ```

  Creates an ellipse annulus section graphic using the specified drawing properties.

  Parameters

item	The ellipse annulus section.
props	The drawing properties.

Public Member Functions

item

```
const ccEllipseAnnulusSection& item() const;
ccEllipseAnnulusSection& item();
```

- ```
  const ccEllipseAnnulusSection& item() const;
  ```

  Returns this graphic item.

- ```
  ccEllipseAnnulusSection& item();
  ```

  Returns a non-**const** reference to this graphic item.

drawArrowHead

```
bool drawArrowHead() const;
void drawArrowHead(bool draw, bool forward);
```

- ```
  bool drawArrowHead() const;
  ```

  Gets the arrow head drawing status. If true, arrow heads are drawn. If false, arrow heads are not drawn.

- `void drawArrowHead(bool draw, bool forward);`

  Draws arrow heads in specified direction.

draw	Flag to draw arrows. True, draws arrow heads. False does not draw arrowheads.
forward	True draws arrows in forward direction. False draws arrows in reverse direction.

drawArrowHeadForward

`bool drawArrowHeadForward() const;`

Returns arrow direction. If true, arrows are drawn in forward direction. If false, arrows are drawn in reverse direction.

encloseRect `ccRect encloseRect() const;`

Returns the enclosing rectangle for underlying graphic item shape.

map `ccGraphicPtrh map(const cc2Xform& c) const;`

Returns a copy of this graphic item, mapped by the transformation object *c*.

Parameters

c	The transformation object.

clone `ccGraphicPtrh clone() const;`

Duplicates this object and returns a pointer to the duplicate.

distToPoint `double distToPoint(const cc2Vect& v) const;`

Retrieves the distance from the ellipse annulus section to the specified point.

Parameters

v	Two-dimensional vector specifying the coordinates of the point.

ccGraphicEllipseAnnulusSection

ccGraphicList

```
#include <ch_cvl/glist.h>

class ccGraphicList;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Simple

ccGraphicList is a container class for **ccGraphic** classes and provides an iterable list of graphic items.

Constructors/Destructors

ccGraphicList

```
ccGraphicList();

ccGraphicList(const ccGraphicList& gl);
```

- `ccGraphicList();`

  Creates a graphic list with no items.

- `ccGraphicList(const ccGraphicList& gl);`

  Creates a deep copy of a graphic list. Every item in the list is copied.

  Parameters

  gl The graphic list to copy.

Operators

operator=

```
ccGraphicList& operator=(const ccGraphicList& gl);
```

Parameters

gl The graphic list to assign.

Assigns a deep copy of the graphic list. Every item in the list is assigned.

Public Member Functions

items

```
const cmStd vector<ccGraphicPtrh>& items() const;
cmStd vector<ccGraphicPtrh>& items();
```

- `const cmStd vector<ccGraphicPtrh>& items() const;`

  Returns the items in this graphic list.

- `cmStd vector<ccGraphicPtrh>& items();`

  Returns a non-**const** reference to the items in this graphic list.

append

```
void append(ccGraphicPtrh item, bool deepCopy=true);
void append(const ccGraphicList& gl);
void append(ccGraphicList& gl, bool deepCopy=true);
```

- `void append(ccGraphicPtrh item, bool deepCopy=true);`

  Appends the graphic item to the end of this list.

  Parameters

item	The item to append.
deepCopy	If true appends a deep copy of the item. If false appends only a copy of the pointer handle to the item.

- `void append(const ccGraphicList& gl);`

  Appends a deep copy of all the items in the list to the end of this list.

  Parameters

gl	The graphic list to append.

- `void append(ccGraphicList& gl, bool deepCopy=true);`

  Appends a graphic list to the end of this list.

  Parameters

gl	The graphic list to append.

deepCopy If true appends a deep copy of all the items in the list. If false appends only the pointer handles to the items in the list.

Notes

If *gl* is appended with a *deepCopy* value of false, the list contains only the pointer handles to its items.

reset

```
void reset();
```

Remove and delete all items from this list.

draw

```
void draw(ccUITablet& tablet,
  ccUITablet::Layers layer = ccUITablet::eImageLayer)
  const;
```

Draw the items in this list into the given tablet. Items are added to the tablet in the same order as they exist in the **items()** vector.

Parameters

tablet The tablet to draw the items into.

layer The plane of the tablet to draw into.

sketch

```
ccUISketch sketch(
  ccUITablet::Layers layer = ccUITablet::eImageLayer)
  const;
```

Returns a sketch of the items in this list. Items are added to the sketch in the same order as they exist in the **items()** vector.

Parameters

layer The plane of the tablet to draw into.

ccGraphicPointIcon

```
#include <ch_cvl/glist.h>

class ccGraphicPointIcon : public ccGraphic;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This class provides the functionality of **ccUITablet::drawPointIcon()** for point icon graphic items.

Constructors/Destructors

ccGraphicPointIcon

```
ccGraphicPointIcon();

ccGraphicPointIcon(const ccPoint& pt, const ccColor& clr);

ccGraphicPointIcon(const ccPoint& pt,
  const ccGraphicProps& props);
```

- `ccGraphicPointIcon();`

  Creates a point icon graphic.

- `ccGraphicPointIcon(const ccPoint& pt, const ccColor& clr);`

  Creates a point icon graphic with specified location and color.

  Parameters

pt	The location of the point.
clr	The color.

- ```
  ccGraphicPointIcon(const ccPoint& pt,
    const ccGraphicProps& props);
  ```

  Creates a point icon graphic at the specified location using the specified drawing properties.
  ```

Parameters

pt	The location of the point.
props	The drawing properties.

Public Member Functions

item

```
const ccPoint& item() const;
ccPoint& item();
```

- `const ccPoint& item() const;`

  Returns the location of this point icon graphic item.

- `ccPoint& item();`

  Returns a non-**const** reference to the location of this point icon graphic item.

encloseRect

```
ccRect encloseRect() const;
```

Returns the enclosing rectangle for this point icon graphic item.

map

```
ccGraphicPtrh map(const cc2Xform& c) const;
```

Returns a copy of this point icon graphic item, mapped by the transformation object *c*.

Parameters

c	The transformation object.

clone

```
ccGraphicPtrh clone() const;
```

Duplicates this object and returns a pointer to the duplicate.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Retrieves the distance from the point icon to the specified point.

Parameters

v	Two-dimensional vector specifying the coordinates of the point.

ccGraphicProps

```
#include <dispprop.h>

class ccGraphicProps : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccGraphicProps** class provides an interface for specifying display properties (for example, rendering, line width, and pen attributes) for both static and manipulable shapes. All graphical shapes do not support all attributes and properties.

Constructors/Destructors

ccGraphicProps

```
ccGraphicProps();

ccGraphicProps(const ccColor &penColor,
  c_Int32 penWidth = 1,
  cePenStyle penStyle = ePenStyleSolid,
  bool penFill = false);

virtual ~ccGraphicProps();
```

- `ccGraphicProps();`

  Default constructor. Instantiates a **ccGraphicProps** object with data members initialized to the following values:

Attribute	Value
Pen color	ccColor::cyanColor()
Pen width	1
Pen style	ePenStyleSolid
Pen fill	false
Pen join	ePenJoinRound

Attribute	Value
Pen end cap	ePenEndCapRound
Arrowhead	true
Arrowhead forward	true
Show genPoly vertex	false

- `ccGraphicProps() ccGraphicProps(const ccColor &penColor, c_Int32 penWidth = 1, cePenStyle penStyle = ePenStyleSolid, bool penFill = false);`

  Convert constructor. Instantiates a **ccGraphicProps** object and initializes a minimum set of data members to the following values:

  Parameters

penColor	Color of the pen used for drawing graphics (default = cyan).
penWidth	Width of the pen used for drawing graphics (default = 1 pixel).
penStyle	Style of the pen used for drawing graphics (default = solid).
penFill	Fill property indicating whether to fill the graphic with color upon rendering (default = false).

- `virtual ~ccGraphicProps();`

  Destructor. Destroys a **ccGraphicProps** object.

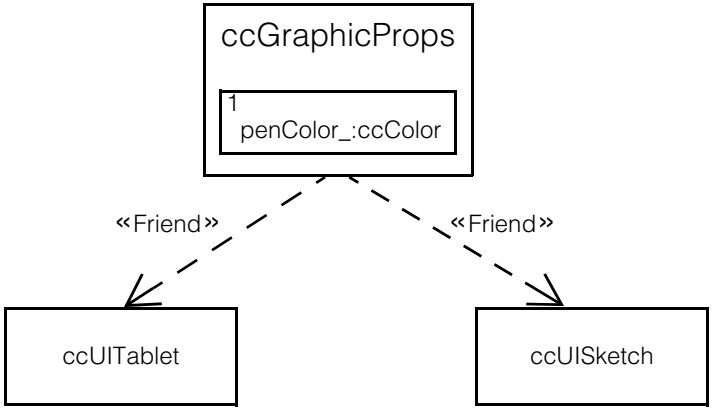


Figure 1. UML Class Diagram of Graphic Properties Class

Operators

operator==

```
bool operator==(const ccGraphicProps &p) const;
```

Returns true if the graphic properties object is equal to another one, false otherwise.

Parameters

p	The other graphics properties object.

operator!=

```
bool operator!=(const ccGraphicProps &p) const;
```

Returns true if the graphic properties object is not equal to another one, false otherwise.

Parameters

p	The other graphics properties object.

Enumerations

cePenStyle

```
enum cePenStyle
```

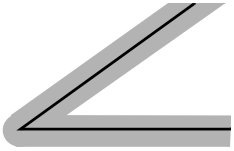
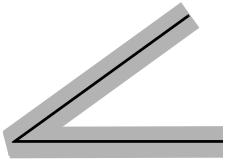
This enumeration specifies the predefined pen styles available for rendering graphics.

Value	Meaning	Example
ePenStyleSolid	Solid, unbroken pixels	
ePenStyleDash	Equally spaced large dashes	
ePenStyleDot	Equally spaced small dashes	
ePenStyleDashDot	Equally alternating large and small dashes	
ePenStyleDashDotDot	Equally alternating large dash and two small dashes	

cePenJoin

`enum cePenJoin`

This enumeration specifies the predefined pen join styles available for rendering graphics.

Value	Meaning	Example
ePenJoinRound	Rounded joins formed by centering circles with diameters equal to the pen width at each of the joins.	
ePenJoinBeveled	Beveled joins formed by filling the corner gaps between each of the two flat-capped lines.	

cePenEndCap

`enum cePenEndCap`

This enumeration specifies the predefined pen end caps available for rendering graphics.

Value	Meaning	Example
ePenEndCapRound	Rounded end caps	
ePenEndCapFlat	Flat end caps abut the end of the line.	
ePenEndCapSquare	Square end caps extend beyond the end of the line.	

Public Member Functions

penColor

```
void penColor(const ccColor &penColor);
const ccColor &penColor() const;
```

- `void penColor(const ccColor &penColor);`

  Sets the pen color.

  Parameters

  penColor — Color used to draw graphics. Must be one of the colors defined by the **ccColor** class.

- `const ccColor &penColor() const;`

  Gets the current pen color.

penWidth

```
c_Int32 penWidth() const;
void penWidth(c_Int32 penWidth);
```

- `c_Int32 penWidth() const;`

  Gets the pen width in pixels.

- `void penWidth(c_Int32 penWidth);`

  Sets the pen width in pixels. The pen width applies to any pen style.

  Parameters

  penWidth — Width of the pen in pixels. Setting *penWidth* to 0 is equivalent to setting it to 1.

  Throws

  ccGraphicProps::BadParams
  If *penWidth* is less than 0

Notes

The default pen width is 1 pixel. The magnification factor of the display does not affect the pen width. A pen width of 0 maps to a 1 pixel wide pen and is used to take advantage of special hardware features when rendering of width 0 is available.

For pen widths greater than 8, the user will not be able to click on the displayed border to select the shape and the cursor will not change to a cross when positioned over the outer or inner perimeter of the fat border. To avoid this problem, call **ccUIManShape::touchDist(float)** to increase the distance at which selection can occur. For a larger pen with, adjust the argument value accordingly. The following is an example of code that increases the touch distance using this method:

```
ccGraphicProps props;
props.penWidth(30);
ccUICircle *uiC = new ccUICircle;
uiC->circle(ccCircle(cc2Vect(350, 350), 100));
uiC->condVisible(true);
uiC->pos(cc2Vect(200, 200));
uiC->condEnabled(true);
uiC->props(props);
uiC->touchDist(15.0);         // Call this to fix the problem
d.addShape(uiC, ccDisplay::eImageCoords);
```

penStyle

```
void penStyle(cePenStyle penStyle);

cePenStyle penStyle() const;
```

- `void penStyle(cePenStyle penStyle);`

Sets the pen rendering style.

Parameters

penStyle	Must be one of the pen styles enumerated by *ccGraphicProps::cePenStyle*.

- `cePenStyle penStyle() const;`

Gets the pen rendering style.

Notes

The default value for pen style is *ccGraphicProps::ePenStyleSolid*.

fill

```
void fill(bool fill);
bool fill() const;
```

- `void fill(bool fill);`

  Sets whether to fill the graphic with color upon rendering.

- `bool fill() const;`

  Gets whether the object is to be filled upon rendering.

  Notes

  The default behavior is to not fill when rendering. The **fill()** methods are implemented only for the graphic shapes **ccEllipse**, **ccRect**, **ccCircle**, and **ccGenRect**.

penJoin

```
void penJoin(cePenJoin penJoin);
cePenJoin penJoin() const;
```

- `void penJoin(cePenJoin penJoin);`

  Sets the pen rendering join shape. Must be one of the shapes enumerated by *ccGraphicProps::cePenJoin*.

- `cePenJoin penJoin() const;`

  Gets the pen rendering join shape.

  Notes

  The default value is *ccGraphicProps::ePenRound*. The pen join shape setting applies only if the pen width is greater than 1.

penEndCap

```
void penEndCap(cePenEndCap penCap);
cePenEndCap penEndCap() const;
```

- `void penEndCap(cePenEndCap penCap);`

  Sets the pen rendering end cap.

  Parameters

penCap	Pen rendering end cap style. Must be one of the end cap styles specified by **cePenEndCap**.

- `cePenEndCap penEndCap() const;`

  Gets the pen rendering end cap.

  Notes

  The default value for the pen rendering end cap is *ccGraphicProps::ePenEndCapRound*. The pen cap setting applies only if the pen width is greater than 1.

  The following is an example of code that sets the pen end cap to different styles:

```
ccDisplayConsole* console = new ccDisplayConsole(
  ccIPair(400,400));

ccPelBuffer<c_UInt8> pelbuf(256, 256);

for (c_Int32 y = 0; y < 256; y++)

    memset(pelbuf.pointToRow(y), y, 256);

console->image(pelbuf, true);

cc2Vect lineSeg_startPoint(0, 0);

cc2Vect lineSeg_endPoint(100, 200);

MessageBox(NULL, cmT("Click to draw lineSeg"),cmT("Example"),
  MB_OK);

ccColor c = ccColor::yellowColor();

ccLineSeg lineSeg(cc2Vect(50,50), cc2Vect(200,50));

ccLineSeg lineSeg1(cc2Vect(50,200),cc2Vect(200,200));

//  style=ePenStyleSolid

ccGraphicProps prop = ccGraphicProps(c, 100,
  ccGraphicProps::cePenStyle(0));

//endCap = ePenEndCapSquare

prop.penEndCap(ccGraphicProps::cePenEndCap(2));

ccGraphicLineSeg gLineSeg(lineSeg, prop);

prop.penEndCap(ccGraphicProps::cePenEndCap(1));

ccGraphicLineSeg gLineSeg1(lineSeg1, prop);

ccGraphicList glist;
```

```
glist.append(new ccGraphicLineSeg(gLineSeg));

glist.append(new ccGraphicLineSeg(gLineSeg1));

ccUITablet tablet;

glist.draw(tablet, ccUITablet::eImageLayer );

console->drawSketch(tablet.sketch(), ccDisplay::eImageCoords);

MessageBox(NULL, cmT("Pan the image or move the message box around
to update the endCapSquare"),cmT("Example"), MB_OK);

delete console;
```

arrowHead

```
void arrowHead(bool enable);

bool arrowHead() const;
```

- `void arrowHead(bool enable);`

  Sets whether arrowhead drawing is enabled for **ccEllipseAnnulusSection** and **cc2Wireframe** objects.

  Parameters

  enable True enables the drawing of arrowheads, false disables it.

  Notes

  The default behavior is for drawing of arrowheads to be enabled. This method applies only to **ccEllipseAnnulusSection** and **cc2Wireframe** objects.

- `bool arrowHead() const;`

  Gets whether arrowhead drawing is enabled for **ccEllipseAnnulusSection** and **cc2Wireframe** objects.

  Notes

  The default behavior is for drawing of arrowheads to be enabled. This method applies only to **ccEllipseAnnulusSection** and **cc2Wireframe** objects.

arrowHeadForward

```
void arrowHeadForward(bool dir);

bool arrowHeadForward() const;
```

- `void arrowHeadForward(bool dir);`

  Sets the direction of the arrow head for **ccEllipseAnnulusSection** objects.

  Parameters

  dir — Direction of the arrow head. True means the arrow faces forward, false means it faces backward.

  Notes

  The default behavior is for arrowheads to be drawn in a forward direction. This method applies only to **ccEllipseAnnulusSection** objects.

- `bool arrowHeadForward() const;`

  Gets the direction of the arrowhead for **ccEllipseAnnulusSection** objects.

  Notes

  The default behavior is for arrowheads to be drawn in a forward direction. This method applies only to **ccEllipseAnnulusSection** objects.

showVertex

```
void showVertex(bool showVertex);

bool showVertex() const;
```

- `void showVertex(bool showVertex);`

  Sets whether to show vertices for a **ccGenPoly** object.

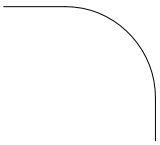
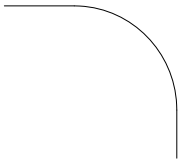
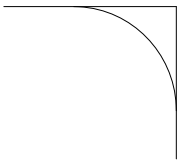
  Parameters

  showVertex — Flag to show or hide the vertices. True means show the vertices, false means hide them.

  Each vertex of a **ccGenPoly** has a corner rounding value, interpreted as follows:

  - A corner rounding value of 0 (the default) causes the corner to be drawn as a sharp intersection of the two segments.
  - A positive corner rounding value causes the corner to be drawn as the radius of a circular arc, or *fillet*, which smoothly blends one segment into the other.
  - A negative corner rounding value tells the pattern matching tools to ignore what would be the circular arc, or *fillet*, if the value were positive.

The **ccGraphicProps::showVertex()** method controls only how vertices with negative corner rounding values are drawn. It is ignored when the rounding value is 0 or positive. Interactions between the corner rounding value of a **ccGenPoly** and **showVertex()** produce the following vertex renderings:

ccGenPoly Corner Rounding Radius	ccGraphicProps:: showVertex()	Rendering
0 (default)	N/A	
Positive	N/A	
Negative	false (default)	
	true	

- `bool showVertex() const;`

  Gets whether to show vertices for a **ccGenPoly** object.

Notes

The default behavior is to hide vertices. The **showVertex** methods are implemented only for **ccGenPoly** objects.

ccGraphicSimple

```
#include <ch_cvl/glist.h>

template <class T>
class ccGraphicSimple : public ccGraphicBuiltin<T>;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

ccGraphicSimple is a template class for **ccGraphicBuiltin** items with no additional parameters. All **ccGraphicSimple** classes are wrapper classes for shapes defined in *shapes.h*. See *Typedefs* on page 1610 for the shapes that are supported by this class.

T Template parameter which specifies the shape of the graphic item. Use one of the **typedef** on page 1610 to specify the class instead of the actual template class name.

Constructors/Destructors

ccGraphicSimple

```
ccGraphicSimple();

ccGraphicSimple(const T& item, const ccColor& clr);

ccGraphicSimple(const T& item,
  const ccGraphicProps& props);
```

- `ccGraphicSimple();`

  Creates an empty simple graphic item.

- `ccGraphicSimple(const T& item, const ccColor& clr);`

  Creates a simple graphic item of specified type and color.

  Parameters

  item The graphic item.

  clr The color.

- ```
  ccGraphicSimple(const T& item,
    const ccGraphicProps& props);
  ```

  Creates a simple graphic of the specified type using the specified drawing properties.

  Parameters

item	The graphic item.
props	The drawing properties.

Public Member Functions

map

```
ccGraphicPtrh map(const cc2Xform& c) const;
```

Returns a copy of this graphic item, mapped by the transformation object *c*.

Parameters

c	The transformation object.

clone

```
ccGraphicPtrh clone() const;
```

Duplicates this object and returns a pointer to the duplicate.

Typedefs

Typedef aliases add Graphic to the class names they represent in *shapes.h*. For example, **ccGraphicPoint** is the synonym for class **ccPoint**.

ccGraphicPoint

```
typedef ccGraphicSimple<ccPoint> ccGraphicPoint;
```

ccGraphic2Point

```
typedef ccGraphicSimple<cc2Point> ccGraphic2Point;
```

ccGraphicLineSeg

```
typedef ccGraphicSimple<ccLineSeg> ccGraphicLineSeg;
```
```

ccGraphicLine
```
typedef ccGraphicSimple<ccLine> ccGraphicLine;
```

ccGraphicFLine
```
typedef ccGraphicSimple<ccFLine> ccGraphicFLine;
```

ccGraphicEllipseArc2
```
typedef ccGraphicSimple<ccEllipseArc2>
                                        ccGraphicEllipseArc2;
```

ccGraphicGenAnnulus
```
typedef ccGraphicSimple<ccGenAnnulus> ccGraphicGenAnnulus;
```

ccGraphicCoordAxes
```
typedef ccGraphicSimple<ccCoordAxes> ccGraphicCoordAxes;
```

ccGraphicPointSet
```
typedef ccGraphicSimple<ccPointSet> ccGraphicPointSet;
```

ccGraphicAffineRectangle
```
typedef ccGraphicSimple<ccAffineRectangle>
  ccGraphicAffineRectangle;
```

ccGraphicPolyline
```
typedef ccGraphicSimple<ccPolyline> ccGraphicPolyline;
```

ccGraphicGenPoly
```
typedef ccGraphicSimple<ccGenPoly> ccGraphicGenPoly;
```

ccGraphic2Wireframe
```
typedef ccGraphicSimple<cc2Wireframe> ccGraphic2Wireframe;
```

ccGraphicBezierCurve
```
typedef ccGraphicSimple<ccBezierCurve>
ccGraphicBezierCurve;
```

ccGraphicDeBoorSpline
```
typedef ccGraphicSimple<ccDeBoorSpline>
ccGraphicDeBoorSpline;
```

ccGraphicInterpSpline
```
typedef ccGraphicSimple<ccInterpSpline>
ccGraphicInterpSpline;
```

ccGraphicHermiteSpline

```
typedef ccGraphicSimple<ccHermiteSpline>
ccGraphicHermiteSpline;
```

Deprecated Members

ccGraphicEllipseArc

```
typedef ccGraphicSimple<ccEllipseArc> ccGraphicEllipseArc;
```

ccGraphicEllipseArc is deprecated. Use **ccGraphicEllipseArc2** above.

ccGraphicPolygon

```
typedef ccGraphicSimple<ccPolygon> ccGraphicPolygon;
```

ccGraphicText

```
#include <ch_cvl/glist.h>

class ccGraphicText : public ccGraphic;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This class provides text strings for graphic items.

Constructors/Destructors

ccGraphicText

```
ccGraphicText();

ccGraphicText(const ccCvlString& str,
  const ccPoint& location, const ccColor& ink,
  const ccColor& backgr, const ccUIFormat& fmt,
  const cc2Vect& oset = cc2Vect(0,0));
```

- `ccGraphicText();`

  Creates an empty graphic text item.

- ```
  ccGraphicText(const ccCvlString& str,
    const ccPoint& location, const ccColor& ink,
    const ccColor& backgr, const ccUIFormat& fmt,
    const cc2Vect& oset = cc2Vect(0,0));
  ```

  Creates a graphic text item with specified text string, location, color, background color, offset, and format.

  Parameters

str	The text string.
location	The text location.
ink	The text color.
backgr	The text background color.
  ```

fmt The text format.

oset The text offset.

Public Member Functions

item

```
const ccCvlString& item() const;
ccCvlString& item();
```

- `const ccCvlString& item() const;`

  Returns the text string and properties for the graphic item.

- `ccCvlString& item();`

  Returns a non-const reference to the text string and properties for the graphic item.

offset

```
cc2Vect offset() const;
void offset(const cc2Vect& oset);
```

- `cc2Vect offset() const;`

  Returns the new offset to original point to draw the label.

- `void offset(const cc2Vect& oset);`

  Sets the new offset to original point to draw the label.

  Parameters

  oset The size of the offset.

location

```
ccPoint location() const;
void location(const ccPoint& loc);
```

- `ccPoint location() const;`

  Gets the location specifying where the label is to be drawn.

- `void location(const ccPoint& loc);`

  Sets the location specifying where the label is to be drawn.

  Parameters

  loc The location.

backgroundColor

```
ccColor backgroundColor() const;
void backgroundColor(const ccColor& backgr);
```

- `ccColor backgroundColor() const;`

  Retrieves the color shown in the background of the label. The color is returned as an object of type **ccColor**.

- `void backgroundColor(const ccColor& backgr);`

  Sets the color displayed in the background of the label.

  Parameters

  backgr The background color.

  Notes

  Specify the background color using one of the static member functions defined in **ccColor**. For example, to specify a green background, use **ccColor::greenColor()**.

format

```
ccUIFormat format() const;
void format(const ccUIFormat& fmt);
```

- `ccUIFormat format() const;`

  Retrieves the format of the label.

  Notes

  The returned format object specifies the font and alignment of the text label. See **ccUIFormat** for more information.

- `void format(const ccUIFormat& fmt);`

  Sest the format of the text label.

Parameters

fmt The format. This object of type **ccUIFormat** specifies the font and alignment of the label. See **ccUIFormat** for more information.

map

```
ccGraphicPtrh map(const cc2Xform& c) const;
```

Returns a copy of this graphic item, mapped by the transformation object *c*.

Parameters

c The transformation object.

clone

```
ccGraphicPtrh clone() const;
```

Duplicates this graphic object and returns a pointer to the duplicate.

ccGraphicWithFill

```
#include <ch_cvl/glist.h>

template<class T, class RT>
class ccGraphicWithFill : public ccGraphicBuiltin<T>;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Yes

ccGraphicWithFill is a template wrapper class for the **ccGraphicBuiltin** graphic items with a fill parameter, **ccRect**, **ccCircle**, **ccEllipse2**, and **ccGenRect**.

T, *RT* — Template parameters which specify the shape of the graphic item. Use one of the **typedef**s on page 1619 to specify the class instead of the actual template class name.

Constructors/Destructors

ccGraphicWithFill

```
ccGraphicWithFill();

ccGraphicWithFill(const T& item, const ccColor& clr,
  bool fill = false);

ccGraphicWithFill(const T& item,
  const ccGraphicProps& props);
```

- `ccGraphicWithFill();`

  Creates an empty graphic with fill object.

- `ccGraphicWithFill(const T& item, const ccColor& clr, bool fill = false);`

  Creates a graphic with fill object of the specified type, color, and fill.

  Parameters

  item — The graphic item.

clr The color.

fill The fill property. True fills the graphic with the specified color. False does not fill the graphic (in other words, the shape is transparent).

- `ccGraphicWithFill(const T& item, const ccGraphicProps& props);`

Creates a graphic with fill object of the specified type using the specified drawing properties.

Parameters

item The graphic item.

props The drawing properties.

---

fill

```
bool fill() const;
void fill(bool f);
```

- `bool fill() const;`

Gets the fill property of this graphic item. If true, shape is filled. If false**,** shape is transparent.

- `void fill(bool f);`

Sets the fill property of this graphic item.

Parameters

f The fill property. True, fills shape with a color. False, does not fill shape. (The shape is transparent).

map

```
ccGraphicPtrh map(const cc2Xform& c) const;
```

Returns a copy of this graphic item, mapped by the transformation object *c*.

Parameters

c The transformation object *c*.

clone

```
ccGraphicPtrh clone() const;
```

Duplicates this graphic object and returns a pointer to the duplicate.

Typedefs

ccGraphicRect

```
typedef ccGraphicWithFill<ccRect,ccGenRect> ccGraphicRect;
```

ccGraphicCircle

```
typedef ccGraphicWithFill<ccCircle,ccEllipse2>
  ccGraphicCircle;
```

ccGraphicAnnulus

```
typedef ccGraphicWithFill<ccAnnulus,ccGenAnnulus>
  ccGraphicAnnulus;
```

ccGraphicEllipse2

```
typedef ccGraphicWithFill<ccEllipse2,ccEllipse2>
  ccGraphicEllipse2;
```

ccGraphicEllipseAnnulus

```
typedef
ccGraphicWithFill<ccEllipseAnnulus,ccEllipseAnnulus>
  ccGraphicEllipseAnnulus;
```

ccGraphicGenRect

```
typedef ccGraphicWithFill<ccGenRect,ccGenRect>
  ccGraphicGenRect;
```

ccGraphicGenAnnulusWithFill

```
typedef ccGraphicWithFill<ccGenAnnulus,ccGenAnnulus>
  ccGraphicGenAnnulusWithFill;
```

Deprecated Members

ccGraphicEllipse

```
typedef ccGraphicWithFill<ccEllipse,ccEllipse>
  ccGraphicEllipse;
```

ccGraphicEllipse is deprecated. Use **ccGraphicEllipse2** above.

ccGreyAcqFifo

```
#include <ch_cvl/acq.h>

class ccGreyAcqFifo : public ccAcqFifo;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This class describes acquisition FIFO queues that return grey-scale images. There is no need for you to create objects of this type in your program.

Notes

It is recommended that you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()**. It returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

Constructors/Destructors

ccGreyAcqFifo

```
ccGreyAcqFifo(const ccVideoFormat& vf, ccFrameGrabber& fg);
```

Creates an idle acquisition FIFO suitable for capturing grey-scale images of the format *vf* from the frame grabber *fg*.

Parameters

vf	The video format of the images to be acquired.
fg	The frame grabber to use to acquire images.

Public Member Functions

baseComplete

```
virtual PelBuffer* baseComplete (ccAcqFailure* result = 0,
  c_UInt32* appTag = 0, bool makeLocal = true,
  double maxWait = HUGE_VAL, bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Returns a pointer to the image of the oldest outstanding acquisition and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one. Returns null if the acquisition failed or if the *maxWait* period elapsed.

Parameters

result — If not null, **result* is set to a description of why the acquisition failed. See **ccAcqFailure** on page 209.

appTag — If not null, **appTag* is set to the value that was passed to **start()**. If the acquisition failed because of too many outstanding acquisitions, or because acquisition was incomplete below then the returned **appTag* value is undefined. For trigger models that do not use **start()** to initiate the acquisition, this value is undefined.

makeLocal — If true, **baseComplete()** forces the image to be local (see **ccPelRoot::move()**). Otherwise, the image remains where it was acquired. For example, if your frame grabber (**ccFrameGrabber**) is also an accelerator (**ccAccelerator**), you might want to set *makeLocal* to false to leave the image on the accelerator to be processed there.

maxWait — The maximum number of seconds to wait for a complete acquisition to become available. The special value *HUGE_VAL* means to wait indefinitely. If the *maxWait* period elapses, **result->isIncomplete()** returns true.

autoStart — If *autoStart* is true and the selected trigger model associated with this FIFO allows **start()** to be invoked, then **start()** is invoked automatically when an acquisition completes. In this case, **start()** is invoked with an *appTag* of zero. If the acquisition is incomplete (**result->isIncomplete()** returns true) or if this function throws, **start()** is not invoked automatically.

startReqStatus — If not NULL, returns whether the acquisition request was processed completely or not. See **ceStartReqStatus** on page 218.

Throws

ccPel::BadWindow

You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.

complete

```
PelBuffer complete (ccAcqFailure* result = 0,
  c_UInt32* appTag = 0, bool makeLocal = true,
  double maxWait = HUGE_VAL, bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Returns the image of the oldest outstanding acquisition and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one. Returns null if the acquisition failed.

Parameters

result — If not null, **result* is set to a description of why the acquisition failed. See **ccAcqFailure** on page 209.

appTag — If not null, **appTag* is set to the value that was passed to **start()**. If the acquisition failed because of too many outstanding acquisitions, or because acquisition was incomplete below then the returned **appTag* value is undefined. For trigger models that do not use **start()** to initiate the acquisition, this value is undefined.

makeLocal — If true, **baseComplete()** forces the image to be local (see **ccPelRoot::move()**). Otherwise, the image remains where it was acquired. For example, if your frame grabber (**ccFrameGrabber**) is also an accelerator (**ccAccelerator**), you might want to set *makeLocal* to false to leave the image on the accelerator to be processed there.

maxWait — The maximum number of seconds to wait for a complete acquisition to become available. The special value *HUGE_VAL* means to wait indefinitely. If the *maxWait* period elapses, **result->isIncomplete()** returns true.

autoStart — If *autoStart* is true and the selected trigger model associated with this FIFO allows **start()** to be invoked, then **start()** is invoked automatically when an acquisition completes. In this case, **start()** is invoked with an *appTag* of zero. If the acquisition is incomplete (**result->isIncomplete()** returns true) or if this function throws, **start()** is not invoked automatically.

startReqStatus — If not NULL, returns whether the acquisition request was processed completely or not. See **ceStartReqStatus** on page 218.

Throws

ccPel::BadWindow — You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.

Typedefs

PelBuffer `typedef ccPelBuffer<c_UInt8> PelBuffer;`

ccGreyVideoFormat

```
#include <ch_cvl/vidfmt.h>

class ccGreyVideoFormat : public ccVideoFormat;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This is an abstract class that generates acquisition FIFOs of type **ccGreyAcqFifo**.

The acquisition software creates a single object of class **ccStdVideoFormat** (see page 3009) for each camera type and image size. You use these video format objects to create an acquisition FIFO that is appropriate for the camera and frame grabber that your application uses.

To get a video format, you use **ccStdVideoFormat::getFormat()** described on page 3012.

Notes

It is recommended that you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()**. It returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

There is no need for you to use this class directly in your code. Use the **ccStdVideoFormat** class for all of your format programming.

Constructors/Destructors

The acquisition software creates one object for each supported video format. You do not create video formats yourself.

Public Member Functions

newAcqFifo

```
ccGreyAcqFifo* newAcqFifo(ccFrameGrabber& fg);
```

Creates and returns a new acquisition FIFO suitable for acquiring images of this format from the frame grabber *fg*.

Parameters

fg	The frame grabber to use to acquire images

Throws

ccVideoFormat::NotSupported
The specified frame grabber does not support this video format.

ccGridCalibParams

```
#include <ch_cvl/calib.h>

class ccGridCalibParams: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the parameters you use to control the operation of the Calibration tool.

Constructors/Destructors

ccGridCalibParams

```
ccGridCalibParams(double gridPitchX, double gridPitchY,
  bool isLeftHanded = true,
  ccCalibDefs::CalibType calibType =
  ccCalibDefs::eGridLinear,
  ccCalibDefs::Polarity polarity =
  ccCalibDefs::eAutoDetect);
```

Constructs a **ccGridCalibParams** with the given parameters. The **ccGridCalibParams** will consider the grid angle of the input image.

Parameters

gridPitchX	The spacing between dots in the x-axis, in client units. *gridPitchX* must be greater than 0.
gridPitchY	The spacing between dots in the y-axis, in client units. *gridPitchY* must be greater than 0.
isLeftHanded	Specify true for this parameter to specify a left-handed client coordinate system.
calibType	The calibration type.
polarity	The polarity of the input image. *polarity* must be set to one of the following values:

ccCalibDefs::eDarkOnLight
ccCalibDefs::eLightOnDark
ccCalibDefs::eAutoDetect

Public Member Functions

gridPitchX

```
double gridPitchX() const;
void gridPitchX(double pitchX) const;
```

- `double gridPitchX() const;`

  Returns the grid pitch in the x-axis in client units.

- `void gridPitchX(double pitchX) const;`

  Sets the grid pitch in the x-axis to *pitchX* in client units.

  Parameters

pitchX	The value the grid pitch in the x-axis is set to

gridPitchY

```
double gridPitchY() const;
void gridPitchY(double pitchY) const;
```

- `double gridPitchY() const;`

  Returns the grid pitch in the y-axis in client units.

- `void gridPitchY(double pitchY) const;`

  Sets the grid pitch in the y-axis to *pitchY* in client units.

  Parameters

pitchY	The value the grid pitch in the y-axis is set to

isLeftHanded

```
bool isLeftHanded() const;
void isLeftHanded(bool hand) const;
```

- `bool isLeftHanded() const;`

  Returns the handedness of the client coordinate system.

- `void isLeftHanded(bool hand) const;`

  Select the handedness of the client coordinate system.

  Parameters

hand	Set to true to select a left handed client coordinate system. Set to false to select a right handed client coordinate system.

calibType

```
ccCalibDefs::CalibType calibType() const;
void calibType(ccCalibDefs::CalibType ctype) const;
```

- `ccCalibDefs::CalibType calibType() const;`

  Returns the calibration type.

- `void calibType(ccCalibDefs::CalibType ctype) const;`

  Sets the calibration type to *ctype*.

  Parameters

ctype	The calibration type this calibration is set to

polarity

```
ccCalibDefs::Polarity polarity() const;
void polarity(ccCalibDefs::Polarity pol) const;
```

- `ccCalibDefs::Polarity polarity() const;`

  Returns the polarity you specified for the input image.

- `void polarity(ccCalibDefs::Polarity pol) const;`

  Set the polarity of the input image to *pol*.

  Parameters

pol	The polarity the input image is set to.

ignoreGridAngle

```
bool ignoreGridAngle() const;
void ignoreGridAngle(bool flag);
```

- `bool ignoreGridAngle() const;`

  Returns true if this **ccGridCalibParams** is configured to ignore any angle between the image coordinate system and the calibration grid. Returns false if this **ccGridCalibParams** is configured to include any angle in the calibration transformation.

- `void ignoreGridAngle(bool flag);`

  Controls whether this **ccGridCalibParams** ignores any angle between the image coordinate system and the calibration grid.

  Parameters

flag	Set to *true* to ignore any angle between the image coordinate system and the calibration grid. Set to *false* to include any angle in the calibration transformation.

ccGridCalibResults

```
#include <ch_cvl/calib.h>

class ccGridCalibResults: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class contains the results of applying the Calibration tool to an input image.

Constructors/Destructors

ccGridCalibResults

```
ccGridCalibResults();
```

Constructs a **ccGridCalibResults** with zero-length vector data members and identity transformations.

Public Member Functions

isMarkerFound

```
bool isMarkerFound() const;
```

Returns true if the most recent call to **cfCalibrationRun()** was able to locate the marker rectangles, false if it could not.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

markerPointInImage

```
cc2Vect markerPointInImage() const;
```

Returns the location of the anchor point (specified as the point where lines drawn through the centers of the marker rectangles intersect) in image coordinates.

Throws

ccCalibDefs::MarkerNotFound
One or both of the marker rectangles were not found.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

clientFromImageXformBase

```
cc2XformBasePtrh clientFromImageXformBase(
  cc2Vect pointInImageSpace = cc2Vect(0,0),
  cc2Vect pointInClientSpace = cc2Vect(0,0)) const;
```

Returns the pointer handle to the transformation that maps image coordinates to client coordinates. This function returns the full transformation, which will be either linear or polynomial depending on the requested calibration type. The optional arguments are used to correctly translate the client coordinate system with respect to the image coordinate system. The points *pointInImageSpace* and *pointInClientSpace* should correspond to the same real-world feature.

For example, to obtain the pointer handle to the transformation that places the intersection of marker rectangles to the client coordinate location 0.56, 0.87, you would make the following call

```
cc2XformBasePtrh myXform =
  myCalibResults.clientFromImageXformBase(
        myCalibResults.markerPointInImage(), // Image point
        cc2Vect(0.56, 0.87));               // Client point
```

Parameters

pointInImageSpace

A point in image coordinates

pointInClientSpace

A point in client coordinates that corresponds to *pointInImageSpace*

Notes

The previous call to **cfCalibrationRun()** must have been successful.

clientFromImageXform

```
cc2Xform clientFromImageXform(
  cc2Vect pointInImageSpace = cc2Vect(0,0),
  cc2Vect pointInClientSpace = cc2Vect(0,0)) const;
```

Returns the **cc2Xform** that maps image coordinates to client coordinates. This function returns the best linear fit, regardless of whether the requested calibration type is linear or polynomial. The optional arguments are used to correctly translate the client

coordinate system with respect to the image coordinate system. The points *pointInImageSpace* and *pointInClientSpace* should correspond to the same real-world feature.

For example, to obtain the **cc2Xform** to the transformation that places the intersection of marker rectangles to the client coordinate location 0.56, 0.87, you would make the following call:

```
cc2Xform myXform =
  myCalibResults.clientFromImageXform(
        myCalibResults.markerPointInImage(), // Image point
        cc2Vect(0.56, 0.87));                // Client point
```

Parameters

pointInImageSpace

A point in image coordinates

pointInClientSpace

A point in client coordinates that corresponds to *pointInImageSpace*

Notes

The previous call to **cfCalibrationRun()** must have been successful.

isLinear

```
bool isLinear() const;
```

Returns true if this object does not contain a polynomial calibration.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

maximumResidual

```
double maximumResidual() const;
```

Returns the maximum residual error in client coordinates.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

averageResidual

```
double averageResidual() const;
```

Returns the average residual error across all grid points in client coordinates.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

residuals

```
cmStd vector<cc2Vect> residuals(
  bool isClient = true) const;
```

Returns the residual error for each grid point in client coordinate system units (if *isClient* is true), or image coordinate units (if *isClient* is false). The ordering of these points in the returned vector is the same as that in the vector returned by **imagePoints()**.

Parameters

isClient — If true, return the residual error in client coordinate system units. If false, return the residual error in image coordinate system units.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

imagePoints

```
cmStd vector<cc2Vect> imagePoints() const;
```

Returns a vector containing the location of the center of each circle in the input image in image coordinates.

Notes

The previous call to **cfCalibrationRun()** must have been successful.

edgeImage

```
ccPelBuffer<c_UInt8> edgeImage() const;
```

Returns a binary edge image showing all the edges detected by the tool. The returned image will have non-edge pixels set to 0 and edge pixels set to 255.

Notes

This member function is provided only for debugging purposes and may not be available in future releases.

This is a diagnostic function; it can be called even if the previous call to **cfCalibrationRun()** was not successful.

segmentedImage

```
ccPelBuffer<c_UInt8> segmentedImage() const;
```

Returns the near-binary segmented image corresponding to the input image. Object pixels are set to 255, background pixels are set to 0 and the intermediate level pixels near the edge are set to a level between those values.

Notes

This member function is provided only for debugging purposes and may not be available in future releases.

This is a diagnostic function; it can be called even if the previous call to **cfCalibrationRun()** was not successful.

threshold

```
c_UInt8 threshold() const;
```

Returns the threshold value computed by **cfCalibrationRun()** to segment the image into grid dots and background.

Notes

This member function is provided only for debugging purposes and may not be available in future releases.

This is a diagnostic function; it can be called even if the previous call to **cfCalibrationRun()** was not successful.

polarity

```
ccCalibDefs::Polarity polarity() const;
```

Returns the actual polarity used by the most recent call to **cfCalibrationRun()**. This may not be the same value you specified. This function returns one of the following values:

ccCalibDefs::eDarkOnLight
ccCalibDefs::eLightOnDark

ccGUI

```
#include <ch_cvl/gui.h>

class ccGUI;
```

A name space that holds constants for console window styles.

Constants

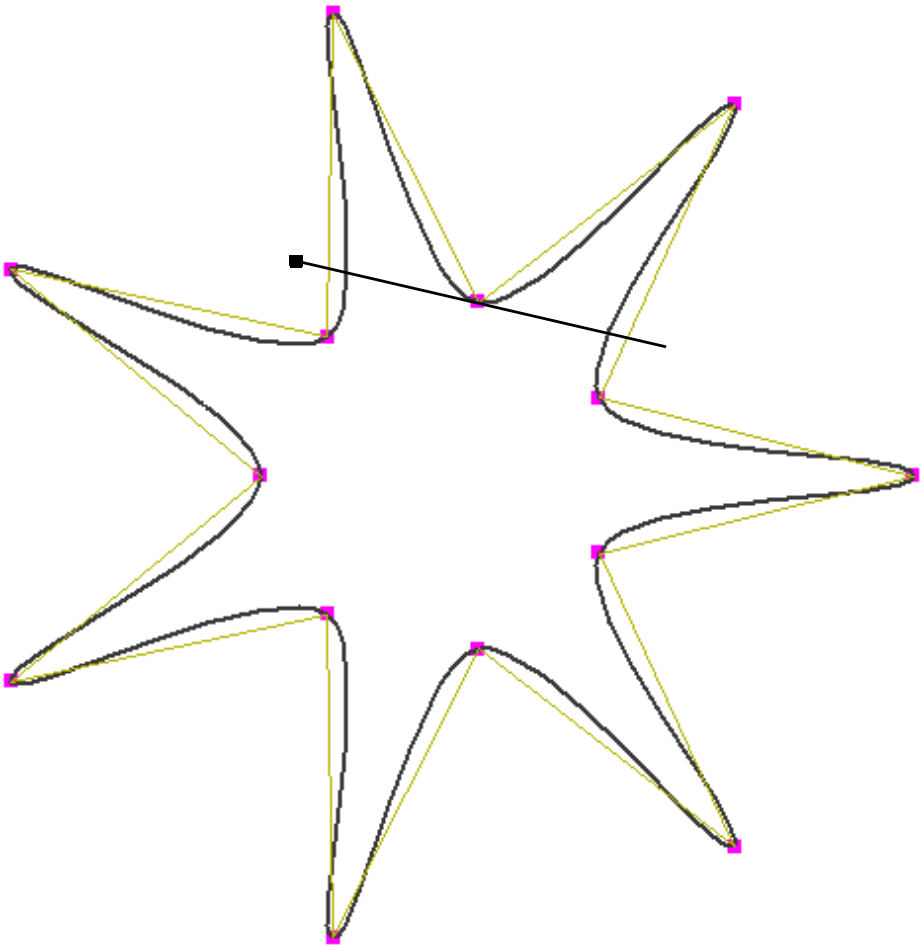
Value	Meaning
kStyleCreateNormal	Creates a visible window.
kStyleCreateMinimized	Creates a minimized window.
kStyleCreateMaximized	Creates a maximized window.
kStyleCreateGUI	Causes a GUI window to be created for the console. If this value is not specified, then output will be sent to **cout**, if it is available.
kStyleCreateNow	Creates and displays the window on construction.
kStyleCreateOnOutput	Creates the window but does not display it until output is sent to the window.
kStyleCloseIsHide	Hide (but do not close) the window when the user clicks the close box.
kStyleNoMessageLoop	Reserved for Cognex use only; do not use.

ccHermiteSpline

```
#include <ch_cvl/spline.h>

class ccHermiteSpline : public ccCubicSpline;
```

The **ccHermiteSpline** class is an implementation of 2D cubic C^1 Hermite interpolation splines. These are another type of cubic interpolation spline that offer additional control knobs. Besides specifying the 2D spline control points that the spline must interpolate, you can also specify the parametric tangent vector of the spline at each control point. Hermite interpolation refers to simultaneous interpolation of point and derivative data.



The above figure shows an example of a Hermite spline with a tangent vector shown at one of the control points.

Properties of Hermite Splines

Note that Hermite splines are not B-Splines, which do not offer control of tangent vectors at internal junction points. The price for the extra control is reduced differentiability at the junctions. In general, the curve is only C^1 at the spline control points; it is infinitely differentiable everywhere else.

The minimum number of spline control points needed to specify a non-empty **ccHermiteSpline** is two, regardless of whether the spline is open or closed. In other words, both open and closed **ccHermiteSpline**s are empty if they have fewer than two control points.

Default Tangent Vectors

When constructing a **ccHermiteSpline** from a list of control points, or when inserting or adding control points, you do not need to provide explicit tangent vectors. If not provided, reasonable tangent vectors are automatically computed using a simple heuristic. If there is only a single control point, its default tangent vector is the zero vector. Otherwise, for *n* control points the default tangent vector for control point *i* is defined in terms of the vector difference between a pair of control points. For open splines, the rules are shown in the following table.

Note In the table, cp[*i*] means controlPoint[*i*] and int[*i*] means interval[*i*].

Value of *i*	Rule
If *i* is equal to zero	(cp[1] - cp[0]) / int[0]
If *i* is greater than zero and less than *n* minus one:	(cp[*i*+1] - cp[*i*-1]) / (int[*i*-1] + int[*i*])
If *i* is equal to *n* minus one:	(cp[*n*-1] - cp[*n*-2]) / int[*n*-2]

Closed splines use the middle rule above in all cases, with modulo *n* indexing. If the denominators of any of the above expressions are zero, the default tangent is set to the zero vector.

Note For in-depth information on the theory and use of 2D cubic Bezier curves and splines, see any textbook on the subject, such as *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, 1990, ISBN 0-12-249051-7.

Note All methods defined for this class leave the **ccHermiteSpline** object unchanged when they throw an error.

Constructors/Destructors

ccHermiteSpline

```
explicit
ccHermiteSpline(bool isClosed = false,
  IntervaleMode intervalMode = eUniform);

explicit
ccHermiteSpline(const cmStd vector<cc2Vect> &ctrlPts,
  bool isClosed = false,
  IntervalMode intervalMode = eUniform);

ccHermiteSpline(const cmStd vector<cc2Vect> &ctrlPts,
  const cmStd vector<cc2Vect> &tangents,
  bool isClosed = false,
  IntervalMode intervalMode = eUniform);
```

- ```
  explicit
  ccHermiteSpline(bool isClosed = false,
    IntervalMode intervalMode = eUniform);
  ```

  Constructs an empty Hermite interpolation spline with no control points, with the given open/closed state, and the given interval mode. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

isClosed	The open/closed state (default = open).
intervalMode	The interval mode. Must be one of the following: *ccCubicSpline::eUniform (default)* *ccCubicSpline:eChordLength* *ccCubicSpline::eCentripetal* *ccCubicSpline::eFixed*

- ```
  explicit
  ccHermiteSpline(const cmStd vector<cc2Vect> &ctrlPts,
    bool isClosed = false,
    IntervalMode intervalMode = eUniform);
  ```

  Constructs a Hermite interpolation spline with the given control points, open/closed state, and interval mode. The tangents of all control points are initialized to default values. See *Default Tangent Vectors* on page 1640. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

ctrlPts	The vector of control points.
isClosed	The open/closed state (default = open).

intervalMode The interval mode. Must be one of the following:
ccCubicSpline::eUniform (default)
ccCubicSpline:eChordLength
ccCubicSpline::eCentripetal
ccCubicSpline::eFixed

- ```
  ccHermiteSpline(const cmStd vector<cc2Vect> &ctrlPts,
    const cmStd vector<cc2Vect> &tangents,
    bool isClosed = false,
    IntervalMode intervalMode = eUniform);
  ```

Constructs a Hermite interpolation spline with the given control points, tangents, open/closed state, and interval mode.

Parameters

ctrlPts The vector of control points.

tangents The vector of tangents.

isClosed The open/closed state (default = open).

intervalMode The interval mode. Must be one of the following:
ccCubicSpline::eUniform (default)
ccCubicSpline:eChordLength
ccCubicSpline::eCentripetal
ccCubicSpline::eFixed

Throws

ccShapesError::BadParams
The *ctrlPts* and *tangents* vectors have different sizes.

Operators

operator==

```
bool operator==(const ccHermiteSpline &rhs) const;
```

Returns true if and only if this **ccHermiteSpline** is equal to *rhs*. Two splines are equal if all of their defining parameters are identical (no tolerance is used).

Parameters

rhs The other **ccHermiteSpline**.

Notes

Splines that describe identical curves are not necessarily equal.
```

operator!=

```
bool operator!=(const ccHermiteSpline &rhs) const;
```

Returns the opposite truth value to **operator==()**.

Parameters

rhs The other **ccHermiteSpline**.

Notes

Splines that describe identical curves are not necessarily equal.

Public Member Functions

controlTangent

```
const cc2Vect &controlTangent(c_Int32 idx) const;

void controlTangent(c_Int32 idx, const cc2Vect &tangent);
```

- ```
  const cc2Vect &controlTangent(c_Int32 idx) const;
  ```

  Gets the tangent of the control point with the given index.

  Parameters

  idx The index.

- ```
  void controlTangent(c_Int32 idx, const cc2Vect &tangent);
  ```

  Sets the tangent of the control point with given index.

  Parameters

  idx The index.

  Throws

  ccShapesError::BadIndex
  idx is less than 0, or greater than or equal to **numControlPoints()**.

controlTangents

```
const cmStd vector<cc2Vect> &controlTangents() const;

void controlTangents(
  const cmStd vector<cc2Vect> &tangents);
```

- ```
  const cmStd vector<cc2Vect> &controlTangents() const;
  ```

  Gets the vector of all control point tangents.
  ```

- ```
  void controlTangents(
    const cmStd vector<cc2Vect> &tangents);
  ```

  Sets the vector of all control point tangents.

  Parameters

  tangents The vector of control point tangents.

  Throws

  ccShapesError::BadParams
  The size of *tangents* is not equal to **numControlPoints()**.

insertControlPoint

```
virtual void insertControlPoint(c_Int32 idx,
  const cc2Vect &point);

void insertControlPoint(c_Int32 idx,
  const cc2Vect &point, const cc2Vect &tangent);
```

- ```
  virtual void insertControlPoint(c_Int32 idx,
    const cc2Vect &point);
  ```

  Inserts a new control point before the point with the given index. The new control point is assigned the default tangent.

  Parameters

  idx The index.

  point The control point.

  Throws

  ccShapesError::BadIndex
  idx is less than 0, or greater than **numControlPoints()** before the insertion.

  Notes

  The new control point is assigned the default tangent. See *Default Tangent Vectors* on page 1640 for details.

- ```
  void insertControlPoint(c_Int32 idx,
    const cc2Vect &point, const cc2Vect &tangent);
  ```

  Inserts a new control point before the point with the given index. Identical to **ccCubicSpline::insertControlPoint()**, except that the new control point is assigned the given tangent rather than the default tangent.
  ```

Parameters

idx The index.

point The control point.

tangent The tangent vector.

Throws

ccShapesError::BadIndex
idx is less than 0, or greater than **numControlPoints()** before the insertion.

See **ccCubicSpline::insertControlPoint()** for details.

addControlPoint

```
void addControlPoint(const cc2Vect &point,
  const cc2Vect &tangent);
```

Convenience function. Appends a control point at the end of the sequence of control points and assigns it the given tangent vector. Equivalent to **insertControlPoint(numControlPoints(), point, tangent)**.

Parameters

point The control point.

tangent The tangent vector.

removeControlPoint

```
virtual void removeControlPoint(c_Int32 idx);
```

Removes the control point with the given index.

Parameters

idx The index.

Throws

ccShapesError::BadIndex
idx is less than 0, or greater than or equal to **numControlPoints()** before the removal.

See **ccCubicSpline::removeControlPoint()** for more information.

map

```
ccHermiteSpline map(const cc2Xform& X) const;
```

Returns this **ccHermiteSpline** mapped by *X*.

Parameters

X The transformation object.

Notes

This function maps the control point positions and tangents by X and leaves the intervals unchanged. It may be desirable to invoke **reparameterize()** on the returned spline. See *Mapping Splines* and *Reparameterizing Splines* on page 1112 for details.

Note Several of **ccCubicSpline**'s setter/getter pairs have a virtual setter and a non-virtual getter. The getters must be explicitly exposed in derived classes, or they will be hidden by the setter overrides.

---

isClosed

```
bool isClosed() const;

virtual void isClosed(bool);
```

- `bool isClosed() const;`

  Gets the open/closed state of this **ccHermiteSpline** (true = closed, false = open).

- `virtual void isClosed(bool);`

  Sets the open/closed state of this **ccHermiteSpline**.

  Parameters

  bool True sets this spline to closed, false sets it to open.

  See **ccCubicSpline::isClosed()** for more information.

---

controlPoint

```
cc2Vect controlPoint(c_Int32 idx) const;

virtual void controlPOint(c_Int32 idx,
  const cc2Vect &ctrlPt);
```

- `cc2Vect controlPoint(c_Int32 idx) const;`

  Gets the control point with the given index.

  Parameters

  idx The index.

- ```
  virtual void controlPoint(c_Int32 idx,
    const cc2Vect &ctrlPt);
  ```

  Sets the control point with the given index.
  ```

Parameters

idx The index.

ctrlPt The control point.

Notes

This function leaves the tangent of the control point unchanged.

See **ccCubicSpline::controlPoint()** for more information.

controlPoints

```
const cmStd vector<cc2Vect> &controlPoints() const;

virtual void controlPoints(
  const cmStd vector<cc2Vect> &ctrlPts);
```

- `const cmStd vector<cc2Vect> &controlPoints() const;`

  Gets the entire vector of control points.

- ```
  virtual void controlPoints(
    const cmStd vector<cc2Vect> &ctrlPts);
  ```

  Sets the entire vector of control points.

  Parameters

  ctrlPts The vector of control points.

  Notes

  This function leaves the tangents of existing control points unchanged. If *ctrlPts* has size *n*, the tangents of the first *n* existing control points will be used for the new points. If there are fewer than *n* existing control points, the surplus *ctrlPts* willreceive default tangents. See *Default Tangent Vectors* on page 1640 for details.

  See **ccCubicSpline::controlPoints()** for more information.

reparameterize

```
virtual void reparameterize();
```

Recomputes all intervals based on the current control points and interval mode. See **ccCubicSpline::reparameterize()** for more information.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this Hermite spline.
```

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this **ccHermiteSpline**. See **ccShape::reverse()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this **ccHermiteSpline** mapped by *X*.

Parameters

X	The transformation object.

See **ccShape::mapShape()** for more information.

ccHistoStats

```
#include <ch_cvl/histstat.h>

class ccHistoStats;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class is used to compute statistics for a histogram.

Constructors/Destructors

ccHistoStats

```
ccHistoStats();

ccHistoStats(cmStd vector<c_UInt32>& histogram,
  bool xown = false);
```

- `ccHistoStats();`

  Constructs a **ccHistoStats** with no histogram. No statistics can be computed by this **ccHistoStats** object.

- `ccHistoStats(cmStd vector<c_UInt32>& histogram, bool xown = false);`

  Constructs a **ccHistoStats** using the given histogram. All statistics will be computed for the supplied histogram.

  Parameters

  histogram — The histogram for which to compute statistics. *histogram* must have at least one element.

  xown — If *xown* is true, this **ccHistoStats** takes over ownership of *histogram*. The vector to which you supplied a reference is set to zero size following the execution of this constructor. If *xown* is false, this **ccHistoStats** makes a copy of *histogram* and the vector to which you supplied a reference is not changed.

Throws

ccHistoStats::Overflow
The sum of histogram bins exceeds *ckMaxUInt32*.

Public Member Functions

isValid

```
bool isValid() const;
```

Returns false if this **ccHistoStats** object was default constructed, otherwise returns true.

nSamp

```
c_UInt32 nSamp() const;
```

Returns the total number of samples in the histogram supplied at construction.

This **ccHistoStats** must not have been default constructed.

mean

```
double mean() const;
```

Returns the arithmetic mean of the values in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

mode

```
c_Int32 mode() const;
```

Returns the modal (most common) value in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

median

```
c_Int32 median() const;
```

Returns the median value in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

histMin

```
c_Int32 histMin() const;
```

Returns the minimum value in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

histMax

```
c_Int32 histMax() const;
```

Returns the maximum value in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

inverseCum

```
c_Int32 inverseCum(int percent) const;
```

Returns the value below which a specified percentage of values in the histogram fall. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

Parameters

percent The percentage below which to return the number of values.

Notes

This function clamps the input *percent* values to the range 0 through 100, inclusive.

sDev

```
double sDev() const;
```

Returns the standard deviation of the values in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

var

```
double var() const;
```

Returns the variance of the values in the histogram supplied at construction. Returns 0 if **nSamp()** returns 0.

This **ccHistoStats** must not have been default constructed.

histogram

```
const cmStd vector<c_UInt32> &histogram() const;
```

Returns the histogram given at construction

Deprecated Members

min

```
c_Int32 min() const;
```

This function is deprecated. Use **histMin()** instead for new development.

max `c_Int32 max() const;`

This function is deprecated. Use **histMax()** instead for new development.

Note The **min()** and **max()** functions can be used only when **min** and **max** are not defined in Windows. Prior to CVL 6.1, the only way you could include the *<histstat.h>* header file was to define **NOMINMAX** before including *<windows.h>* or *<afx.h>*. Therefore, all current CVL applications are not affected and can continue to use the **min()** and **max()** functions.

New applications created with CVL 6.1 and later can include *<histstat.h>* and use the **histMin()** and **histMax()** functions without first undefining the Windows **min** and **max** symbols.

ccIDDecodeParams

```
#include <ch_cvl/id.h>

class ccIDDecodeParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the decoding parameters for the ID tool. Individual members are provided for symbology-specific decoding parameters.

Constructors/Destructors

ccIDDecodeParams

```
ccIDDecodeParams();
```

Constructs a **ccIDDecodeParams**. The following symbologies are enabled in a default constructed object:

- Code39,
- Code93,
- Code128,
- Codabar,
- UPC/EAN

ccIDDecodeParams::qualityOption is set to *ccIDDefs::eQualityOptionNone* and the symbology-specific parameter members are set to default-constructed symbology-specific parameter objects.

Public Member Functions

isSymbologyEnabled

```
bool isSymbologyEnabled(ccIDDefs::Symbology symbology)
   const;
```

Returns true if the given symbology is enabled in this object, false otherwise.

Parameters

symbology The symbology.

Throws

ccIDDefs::BadParams

symbology is invalid.

enableSymbology

```
void enableSymbology(ccIDDefs::Symbology symbology,
   bool enable);
```

Enables or disables the specified symbology.

Parameters

symbology The symbology.

enable Specify true to enable the symbology, false to disable it.

Notes

In order to read EAN.UCC Composite code, both symbologies for the linear and Composite need to be enabled. For example, to read EAN.UCC RSS Composite code, both *ccIDDefs::eSymbologyRSS* and *ccIDDEfs::eSymbologyComposite* need to be enabled.

Throws

ccIDDefs::BadParams

symbology is invalid. *symbology* must be a member of **ccIDDefs::Symbology** and it must not be *ccIDDefs::eSymbologyUnknown*.

disableAllSymbologies

```
void disableAllSymbologies();
```

Disables all symbologies in this object.

resetEnabledSymbologies

```
void resetEnabledSymbologies();
```

Enables the symbologies that are enabled for a default-constructed **ccIDDecodeParams** object;

- Code39
- Code93
- Code128
- Codabar

- UPC/EAN

All other symbologies are disabled.

Notes

In order to read EAN.UCC Composite code, both symbologies for the linear and Composite need to be enabled. For example, to read EAN.UCC RSS Composite code, both *ccIDDefs::eSymbologyRSS* and *ccIDDEfs::eSymbologyComposite* need to be enabled.

Throws

ccIDDefs::BadParams

symbology is not a member of **ccIDDefs::Symbology**.

qualityOption

```
c_UInt32 qualityOption() const;

void qualityOption(c_UInt32 qualityOption);
```

- `c_UInt32 qualityOption() const;`

Returns the currently enabled quality options. The returned value is formed by ORing together 1 or more the following values:

ccIDQualityDefs::eQualityOptionNone
ccIDQualityDefs::eQualityOptionBasic

- `void qualityOption(c_UInt32 qualityOption);`

Sets the quality options. The quality options determine which quality measures are computed for a read symbol.

The available options are *ccIDQualityDefs::eQualityOptionNone* (no measures are computed) and *ccIDQualityDefs::eQualityOptionBasic* (the basic options including ANSI print quality are computed).

Parameters

qualityOption The options to compute. This value must be formed by ORing together one or more of the following values:

ccIDQualityDefs::eQualityOptionNone
ccIDQualityDefs::eQualityOptionBasic

Throws

ccIDDefs::BadParams

qualityOption is not a valid bitwise combination of the **ccIDQualityDefs::QualityOption** values.

paramsI2of5

```
const ccSymbologyParamsI2of5& paramsI2of5() const;

void paramsI2of5(
   const ccSymbologyParamsI2of5& paramsI2of5);
```

- `const ccSymbologyParamsI2of5& paramsI2of5() const;`

  Returns the currently specified 2 of 5 decoding parameters object.

- ```
  void paramsI2of5(
     const ccSymbologyParamsI2of5& paramsI2of5);
  ```

  Sets the 2 of 5 decoding parameters object.

  Parameters

  paramsI2of5 The parameter object to set.

paramsCode39

```
const ccSymbologyParamsCode39& paramsCode39() const;

void paramsCode39(
   const ccSymbologyParamsCode39& paramsCode39);
```

- `const ccSymbologyParamsCode39& paramsCode39() const;`

  Returns the currently specified Code 39 decoding parameters object.

- ```
  void paramsCode39(
     const ccSymbologyParamsCode39& paramsCode39);
  ```

  Sets the Code 39 decoding parameters object.

  Parameters

  paramsCode39 The parameter object to set.

paramsCode128

```
const ccSymbologyParamsCode128& paramsCode128() const;

void paramsCode128(
   const ccSymbologyParamsCode128& paramsCode128);
```

- `const ccSymbologyParamsCode128& paramsCode128() const;`

  Returns the currently specified Code 128 decoding parameters object.

- ```
  void paramsCode128(
     const ccSymbologyParamsCode128& paramsCode128);
  ```

  Sets the Code 128 decoding parameters object.

  Parameters
  paramsCode128 The parameter object to set.

paramsUPCEAN

```
const ccSymbologyParamsUPCEAN& paramsUPCEAN() const;

void paramsUPCEAN(
   const ccSymbologyParamsUPCEAN& paramsUPCEAN);
```

- `const ccSymbologyParamsUPCEAN& paramsUPCEAN() const;`

  Returns the currently specified UPC/EAN decoding parameters object.

- ```
  void paramsUPCEAN(
     const ccSymbologyParamsUPCEAN& paramsUPCEAN);
  ```

  Sets the UPC/EAN decoding parameters object.

  Parameters
  paramsUPCEAN The parameter object to set.

paramsCodabar

```
const ccSymbologyParamsCodabar& paramsCodabar() const;

void paramsCodabar(
   const ccSymbologyParamsCodabar& paramsCodabar);
```

- `const ccSymbologyParamsCodabar& paramsCodabar() const;`

  Returns the currently specified Codabar decoding parameters object.

- ```
  void paramsCodabar(
     const ccSymbologyParamsCodabar& paramsCodabar);
  ```

  Sets the Codabar decoding parameters object.

  Parameters
  paramsCodabar The parameter object to set.

paramsCode93

```
const ccSymbologyParamsCode93& paramsCode93() const;

void paramsCode93(
   const ccSymbologyParamsCode93& paramsCode93);
```

- `const ccSymbologyParamsCode93& paramsCode93() const;`

  Returns the currently specified Code 93 decoding parameters object.

- ```
  void paramsCode93(
     const ccSymbologyParamsCode93& paramsCode93);
  ```

  Sets the Code 93 decoding parameters object.

  Parameters
  paramsCode93 The parameter object to set.

paramsPDF417

```
const ccSymbologyParamsPDF417& paramsPDF417() const;

void paramsPDF417(const ccSymbologyParamsPDF417&
paramsPDF417);
```

- `const ccSymbologyParamsPDF417& paramsPDF417() const;`

  Returns the currently specified PDF417 decoding parameters object.

- `void paramsPDF417(const ccSymbologyParamsPDF417& paramsPDF417);`

  Sets the PDF417 decoding parameters object.

  Parameters
  paramsPDF417 The parameter object to set.

paramsRSS

```
const ccSymbologyParamsRSS& paramsRSS() const;

void paramsRSS(const ccSymbologyParamsRSS& paramsRSS);
```

- `const ccSymbologyParamsRSS& paramsRSS() const;`

  Returns the currently specified RSS decoding parameters object.

- `void paramsRSS(const ccSymbologyParamsRSS& paramsRSS);`

  Sets the RSS decoding parameters object.

  Parameters
  paramsRSS The parameter object to set.

paramsComposite

```
const ccSymbologyParamsComposite& paramsComposite() const;

void paramsComposite(
   const ccSymbologyParamsComposite& paramsComposite);
```

- `const ccSymbologyParamsComposite& paramsComposite() const;`

  Returns the currently specified Composite decoding parameters object.

- `void paramsComposite(`
  `const ccSymbologyParamsComposite& paramsComposite);`

  Sets the Composite decoding parameters object.

  Parameters
  paramsComposite
  The parameter object to set.

paramsPostal

```
const ccSymbologyParamsPostal& paramsPostal() const;

void paramsPostal(
   const ccSymbologyParamsPostal& paramsPostal);
```

- `const ccSymbologyParamsPostal& paramsPostal() const;`

  Returns the currently specified Postal decoding parameters object.

- ```
  void paramsPostal(
     const ccSymbologyParamsPostal& paramsPostal);
  ```

  Sets the Postal decoding parameters object.

  Parameters

  paramsPostal The parameter object to set.

Operators

operator==

```
bool operator== (const ccIDDecodeParams& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.
```

ccIDDecodeResult

```
#include <ch_cvl/id.h>

class ccIDDecodeResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the results from decoding a single symbol.

Constructors/Destructors

ccIDDecodeResult

```
ccIDDecodeResult();
```

Constructs a **ccIDDecodeResult** with the following values:

ccIDDecodeResult::symbology set to *ccIDDefs::eSymbologyUnknown*
ccIDDecodeResult::symbologySubtype set to 0
ccIDDecodeResult::symbolIdentifiers set to an empty string
ccIDDecodeResult::hasErrorInfo set to false

Public Member Functions

symbology

```
ccIDDefs::Symbology symbology() const;
```

Returns the type of this symbol. The returned value is a member of **ccIDDefs::Symbology**.

For composite symbologies, when **ccSymbologyParamsComposite::combineResults** is true, this function returns *ccIDDefs::eComposite*.

symbologySubtype

```
c_UInt32 symbologySubtype() const;
```

Returns the symbology subtype of this symbol, if applicable.

Valid for UPC/EAN, RSS and 4-State Postal Codes. Always use together with **symbology()** to determine the actual type of the decoded symbol. A Value 0 indicates that the symbology does not have subtypes.

Notes

The subtype values are defined in the symbology parameters classes (**ccSymbologyParamsRSS**, **ccSymbologyParamsPostal**, **ccSymbologyParamsComposite**, and **ccSymbologyParamsUPCEAN**).

symbolIdentifiers

```
const ccCvlString& symbolIdentifiers() const;
```

Returns a null-terminated string of three ASCII characters conforming to ***ISO/IEC 15424:2000*** *Information technology -- Automatic identification and data capture techniques - Data Carrier Identifiers (including Symbology Identifiers)*.

This string includes a flag character, a code character, and a modifier character. The structure of the symbology identifier is:

```
]cm
```

where

] (ASCII value 93 in accordance with ISO646) represents the symbology identifier flag character

c represents the code character which indicates to the host the bar code symbology of the symbol which has been read, where

A = Code 39,
C = Code 128,
E = UPC/EAN,
I = Interleaved 2 of 5,
G = Code 93,
L = PDF417,
e = RSS,
X = other bar codes.

m represents the modifier character as defined for the symbology in question. Specifically, the identifier character 'X' is used to specify symbologies that are not listed in ISO/IEC 15424:2000 and can have the following modifiers:

]X0 for POSTNET/CEPNet, reader has stripped the check character
]X1 for POSTNET/CEPNet, reader has transmitted the check character
]X2 for PLANET, reader has stripped the check character
]X3 for PLANET, reader has transmitted the check character
]X6 for Australia Post 4-state code
]X8 for UPU 4-State postal code
]X9 for Japan Post 4-State customer barcode
]XA for USPS 4-State postal code
]XB for Pharmacode one-track code

Throws

ccIDDefs::BadString
This is a default-constructed **ccIDDecodeResult**.

decodedMBCString

```
const cmStd vector<c_UInt8>& decodedMBCString() const;
```

Returns the decoded data as a double-null terminated multi-byte character string.

For symbologies that encode 8-bit characters, this function returns a string of single-byte characters. For symbologies that encode both single-byte and double-byte characters, it returns a stream of multibyte characters.

Throws

ccIDDefs::BadString
This is a default-constructed **ccIDDecodeResult**.

decodedString

```
ccCvlString decodedString() const;
```

Returns the decoded string.

This function should only be used for symbols which are known to not contain multibyte characters in the decoded data, and for which the interpretation of the characters in the decoded string is up to the application.

For symbols encoding non-alphanumeric data, use **decodedMBCString()** to obtain the decoded data.

Throws

ccIDDefs::BadString
This is a default-constructed **ccIDDecodeResult**.

decodedElementStream

```
const cmStd vector<c_Int32>& decodedElementStream() const;
```

Gets the uninterpreted stream. Interpretation is symbology-specific.

Throws

ccIDDefs::BadString
This is a default-constructed **ccIDDecodeResult**.

hasErrorInfo

```
bool hasErrorInfo() const;
```

Returns true if **numErrors()**, **numErrorBits()**, **numErrorWords()**, and u**nusedErrorCorrection()** will return meaningful information.

numErrors

```
c_Int32 numErrors() const;
```

Returns the number of erroneous data words encountered while decoding the symbol.

Throws

ccIDDefs::NotComputed
There is no error information for this result (**hasErrorInfo()** returns false).

numErrorBits

```
c_Int32 numErrorBits() const;
```

Returns the number of erroneous bits encountered while decoding the symbol.

Throws

ccIDDefs::NotComputed
There is no error information for this result (**hasErrorInfo()** returns false).

unusedErrorCorrection

```
c_Int32 unusedErrorCorrection() const;
```

Returns the amount of unused error correction in the decoded symbol.

The returned value is in the range of 0 through 100 inclusively to indicate the percentage unused error correction. This only applies to symbologies with strong error correction, such as PDF417.

For symbologies that have reserved codewords for error detection to prevent misreads, this value can be negative if the error detection codewords are used for error correction. A negative value indicates that the number of erasures and errors is beyond the specified limit for correction. The closer a negative value is to zero, the less the probability of getting a misread.

In applications where a high decode rate is desired at the risk of low probability misreads, **isDecoded()** can be used as the sole indication of a successful decode. When a slightly lower but more confident decode rate is preferred, you should check that the **unusedErrorCorrection()** is non-negative before considering the decoded result to be valid.

Throws

ccIDDefs::NotComputed
There is no error information for this result (**hasErrorInfo()** returns false).

Operators

operator==

```
bool operator== (const ccIDDecodeResult& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDDecodeResult

ccIDDefs

```
#include <ch_cvl/id.h>

class ccIDDefs;
```

A name space that holds enumerations, constants, and errors associated with the ID tool.

Enumerations

Symbology

```
enum Symbology;
```

This enumeration defines the symbology types supported by the ID tool. You use this enumeration to tell the tool what symbologies to attempt to decode, and the tool returns the actual symbology for a decoded symbol.

Value	Symbology
eSymbologyUnknown	Unknown.
eCodabar	Codabar
eCode128	Code 128
eCode39	Code 39
eCode93	Code 93
eI2of5	Interleaved 2 of 5
ePharmaCode	Pharmacode
eUPCEAN	UPC / EAN
ePostal	Postal codes (4-state Japan Post, Australia Post, UPU, and USPS; POSTNET, PLANET)
eRSS	RSS (DataBar)
ePDF417	PDF417 stacked code
eComposite	EAN.UCC Composite

Mirrored

```
enum Mirrored;
```

This enumeration defines whether or not the symbol is mirrored.

Value	meaning
eMirroredFalse	Symbol is not mirrored.
eMirroredTrue	Symbol is mirrored.
eMirroredUnknown	Mirroring is unknown.

Polarity

```
enum Polarity;
```

This enumeration defines whether the symbol consists of dark bars on a light background or light symbols on a dark background.

Value	meaning
eDarkOnLight	Dark bars on a light background.
eLightOnDark	Light bars on a dark background.
ePolarityUnknown	Unknown polarity
kDefaultPolarity	The default polarity for newly constructed objects.

OperatingMode

```
enum OperatingMode;
```

This enumeration defines the operating mode for the ID tool.

Value	meaning
eStandard = 1	Standard mode.
e1DMax = 2	1DMax mode.
kDefaultOperatingMode = *eStandard*	The default operating mode.

Notes

The e1DMax operating mode is recommended for dealing with hard-to-read codes. For more information on operating modes, see the ID Tool chapter in the CVL Vision Tools Guide.

FailureCode `enum FailureCode;`

This enumeration defines the failure codes that the ID tool can return. These codes provide specific information about decoding failures.

Value	Meaning
eFailureNotApplicable	No failure codes are available for this symbology.
eFailureFinderPatternNotFound	The finder pattern for this symbol was not found.
eFailureStartStopPatternNotComplete	The start/stop pattern for this symbol is not complete.
eFailureChecksumFailed	The checksum for this symbol is incorrect.
eFailureDegradedSymbol	This symbol contains unrecognized characters or characters recognized with low confidence.
eFailureUnexpectedStringLength	The decoded string length is outside the user specified range.
eFailureTooManyErrorBits	The number of error bits detected in the 2D symbol is beyond the capacity of error correction.
eFailureSymbolVersionNotFound	Version of the symbol could not be determined.
eFailureRowAddressPatternOutOfOrder	The detected Row Address patterns of a 2D result are out of sequence.
eFailureCodeLinesNotFound	The detected Row Address patterns of a 2D result cannot form data rows.
eFailureInternalError	The ID tool experienced an internal error.

DrawMode `enum DrawMode;`

This enumeration defines what type of result graphics to draw.

Value	meaning
eDrawCenter	Draw a cross at the center of the result location.
eDrawBoundary	Draw a boundary around the result location.
kDefaultDrawMode	The default graphics to draw.

ccIDQualityDefs

```
#include <ch_cvl/id.h>

class ccIDQualityDefs
```

Static Functions

numQualityMetrics

```
static c_Int32 numQualityMetrics();
```

Returns the number of quality metrics that are defined in the **QualityMetrics** enumeration.

Enumerations

QualityOption

```
enum QualityOption;
```

This enumeration defines which quality measures to compute.

Value	meaning
eQualityOptionNone	No measures.
eQualityOptionBasic	The basic measures: *eEdgeDetermination*, *eEdgeContrastMin*, *eModulation*, *eDefects*, *eReferenceDecode*, *eScanGrade*, and *eSymbolContrast*.
kDefaultQualityOption	The default quality option.

QualityMetrics

```
enum QualityMetrics;
```

This enumeration defines the quality measures. These measures are formally defined by *ANSI X3.182 - 1990 Bar Code Print Quality*.l

Value	meaning
eOverallGrade	The overall quality measure, from A to F.
eEdgeDetermination	Graded A if the count of global threshold based edges in the scan reflectance profile corresponds to a valid symbology, graded F otherwise.
eReflectMin	The lowest value in the reflectance profile.

Value	meaning
eEdgeContrastMin	Graded A if the minimum reflectance difference between symbol and adjacent background is >= 15% of greyscale range, graded F otherwise. See ANSI X3.182.
eModulation	Minimum edge contrast divided by overall symbol contrast. A: >= 0.7, B: >= 0.6, C: >= 0.5, D: >= 0.4, F: < 0.4. See ANSI X3.182.
eDefects	Maximum element reflectance non-uniformity divided by overall symbol contrast. A: <= 0.15, B: <= 0.20, C: <= 0.25, D: <= 0.30, F: > 0.30. See ANSI X3.182.
eReferenceDecode	Graded A if the ANSI Reference Decode algorithm can decode this symbol, graded F otherwise. See ANSI X3.182.
eDecodability	A measure of how close the symbol characteristics are to the nominal values defined for that symbology. See ANSI X3.182.
eScanGrade	The lowest grade received for any reflectance scan profile based quality measurement. See ANSI X3.182.
eSymbolContrast	Difference between highest and lowest reflectance values in the scan profile, as percentage of greyscale range. A: >= 70%, B: >= 55%, C: >= 40%, D: >= 20%, F: < 20%. See ANSI X3.182.

QualityGrade `enum QualityGrade;`

This enumeration defines the quality grades.

Value	meaning
eGradeNotComputed	The grade for this measure was not computed.
eGradeA	Grade A where A is Excellent and F is Fail.
eGradeB	Grade B where A is Excellent and F is Fail.
eGradeC	Grade C where A is Excellent and F is Fail.
eGradeD	Grade D where A is Excellent and F is Fail.
eGradeF	Grade F where A is Excellent and F is Fail.

ccIDQualityDefs

ccIDQualityResult

```
#include <ch_cvl/id.h>

class ccIDQualityResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the quality measures computed from a single decoded symbol.

Constructors/Destructors

ccIDQualityResult

```
ccIDQualityResult();
```

Constructs this object with all quality measures are set to *ccIDQualityDefs::eGradeNotComputed*.

Public Member Functions

qualityGrade

```
ccIDQualityDefs::QualityGrade qualityGrade(
   ccIDQualityDefs::QualityMetrics metric) const;
```

Returns the grading result for the supplied quality metric. The returned value is a member of the **ccIDQualityDefs::QualityGrade** enumeration.

Parameters

metric The metric to compute. The supplied value must be one of the following values:

ccIDQualityDefs::eOverallGrade
ccIDQualityDefs::eEdgeDetermination
ccIDQualityDefs::eReflectMin
ccIDQualityDefs::eEdgeContrastMin
ccIDQualityDefs::eModulation
ccIDQualityDefs::eDefects
ccIDQualityDefs::eReferenceDecode

ccIDQualityDefs::eDecodability
ccIDQualityDefs::eScanGrade
ccIDQualityDefs::eSymbolContrast

Notes

Returns *ccIDQualityDefs::eGradeNotComputed* if the requested metric was not computed.

Throws

ccIDDefs::BadParams

metric is not a member of **ccIDQualityDefs::QualityMetrics**.

operator==

```
bool operator== (const ccIDQualityResult& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that	The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDResult

```
#include <ch_cvl/id.h>

class ccIDResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing all of the result information from a symbol.

Constructors/Destructors

ccIDResult

```
ccIDResult();
```

Constructs this object with the **ccIDResult::subResults** set to a single default-constructed **ccIDSubResult** object.

Public Member Functions

numSubResults

```
c_Int32 numSubResults() const;
```

Returns the number of subresults in this result.

Notes

For composite symbologies this value can be greater than 1.

subResults

```
const cmStd vector<ccIDSubResult>& subResults() const;
```

Returns the vector of subresults contained in this result.

isFound

```
bool isFound() const;
```

Returns the value of **ccIDSubResult::isFound()** for the first subresult **(subResults(0)**).

isDecoded

```
bool isDecoded() const;
```

Returns the value of **ccIDSubResult::isDecoded()** for the first subresult **(subResults(0)**).

ccIDResult

failureCode

```
ccIDDefs::FailureCode failureCode() const;
```

Returns the value of **ccIDSubResult::failureCode()** for the first subresult **(subResults(0)**).

searchResult

```
const ccIDSearchResult& searchResult() const;
```

Returns the search result (**ccIDSubResult::searchResult())** from the first subresult **(subResults(0)**).

Throws

ccIDDefs::NotComputed
There are no results (**isFound()** is false).

decodeResult

```
const ccIDDecodeResult& decodeResult()const;
```

Returns the decode result (**ccIDSubResult::decodeResult())** from the first subresult **(subResults(0)**).

Throws

ccIDDefs::NotComputed
There is no decode result (**isDecoded()** is false).

qualityResult

```
const ccIDQualityResult& qualityResult() const;
```

Returns the quality result (**ccIDSubResult::qualityResult())** from the first subresult **(subResults(0)**).

Notes

When the symbol was decoded but **ccIDDecodeParams::qualityOption** was set to *ccIDQualityDefs::eQualityOptionNone*, a default constructed **ccIDQualityResult** is returned.

Throws

ccIDDefs::NotComputed
There is no decode result (**isDecoded()** is false).

draw

```
void draw(ccGraphicList& glist, c_UInt32 drawMode) const;
```

Appends graphics for this result to the supplied graphics list. *drawMode* is used to select which graphics are drawn and must be a bitwise combination of the following values:

ccIDDefs::eDrawCenter
ccIDDefs::eDrawBoundary

ccIDDefs::eDrawCenter draws a cross at this result's location; *ccIDDefs::eDrawBoundary* draws the boundary of this result, which is a polygon connecting the corners.

Graphics are drawn in *ccColor::green* if the result was successfully decoded, *ccColor::red* if the result was found but not decoded, and no graphics are drawn if no subresult has **isFound()** equal to true.

Parameters

glist	The graphics list to which to append the graphic.
drawMode	The drawing mode.

Notes

All graphics are drawn in client coordinates.

Operators

operator==

```
bool operator== (const ccIDResult& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that	The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDResult

ccIDResultSet

```
#include <ch_cvl/id.h>

class ccIDResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the results of a single invocation of the ID tool. One result is generated for each symbol found in the image, up to the requested number of symbols to find.

Constructors/Destructors

ccIDResultSet

```
ccIDResultSet();
```

Constructs this object with **time** set to 0.0 and **results** set to an empty vector.

Public Member Functions

numResults

```
c_Int32 numResults() const;
```

Returns the number of results in this result set.

time

```
double time() const;
```

Returns the time (in seconds) used to run the tool and create this result.

results

```
const cmStd vector<ccIDResult>& results() const;
```

Returns a vector containing the results of this result set.

draw `void draw(ccGraphicList& glist, c_UInt32 drawMode) const;`

Appends graphics for this result set to the supplied graphics list. *drawMode* is used to select which graphics are drawn and must be a bitwise combination of the following values:

ccIDDefs::eDrawCenter
ccIDDefs::eDrawBoundary

ccIDDefs::eDrawCenter draws a cross at this result's location; *ccIDDefs::eDrawBoundary* draws the boundary of this result, which is a polygon connecting the corners.

Graphics are drawn in *ccColor::green* if the result was successfully decoded, *ccColor::red* if the result was found but not decoded, and no graphics are drawn if no result in this set has **isFound()** equal to true.

Parameters

glist The graphics list to which to append the graphic.

drawMode The drawing mode.

Notes

All graphics are drawn in client coordinates.

Operators

operator== `bool operator== (const ccIDResultSet& that) const;`

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDSearchParams

```
#include <ch_cvl/id.h>

class ccIDSearchParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the search parameters for the ID tool.

Constructors/Destructors

ccIDSearchParams

```
ccIDSearchParams();
```

Constructs an object with the following default values:

numToFind: 1
isPostalOmniDirectional: false
mirror: *ccIDDefs::eMirroredFalse*
polarity: *ccIDDefs::kDefaultPolarity*
operatingMode: *ccIDDefs::kDefaultOperatingMode*

Public Member Functions

numToFind

```
c_Int32 numToFind() const;

void numToFind(c_Int32 numToFind);
```

- ```
  c_Int32 numToFind() const;
  ```

  Returns the number of symbols to find.

- ```
  void numToFind(c_Int32 numToFind);
  ```

  Sets the number of symbols to be found. The tool returns a result for each symbol that it finds up to the number that you specify.

Parameters
numToFind The number to find.

Throws
ccIDDefs::BadParams
numToFind is less than 1.

mirror

```
ccIDDefs::Mirrored mirror() const;
void mirror(ccIDDefs::Mirrored mirror);
```

- `ccIDDefs::Mirrored mirror() const;`

Returns whether the tool will search for mirrored images. The returned value is one of the following:

ccIDDefs::eMirroredFalse
ccIDDefs::eMirroredTrue
ccIDDefs::eMirroredUnknown

If **mirror()** is *ccIDDefs::eMirroredUnknown*, the tool will search for both mirrored and unmirrored symbols.

- `void mirror(ccIDDefs::Mirrored mirror);`

Determines whether the tool will search for mirrored images. Specifying *ccIDDefs::eMirroredUnknown* causes the tool to search for both mirrored and unmirrored symbols.

Parameters
mirror The mirroring flag. *mirror* must be one of the following values:

ccIDDefs::eMirroredFalse
ccIDDefs::eMirroredTrue
ccIDDefs::eMirroredUnknown

Throws
ccIDDefs::BadParams
mirror is invalid.

Notes

The mirror flag is ignored for linear symbologies.

For postal codes, when the mirror flag is set to *ccIDDefs::eMirroredUnknown*, the tool outputs the better decoded postal code result of either mirrored or non-mirrored. When the mirrored and non-mirrored results have the same evaluation, the tool outputs the non-mirrored result. Cognex recommends that you do not specify *ccIDDefs::eMirroredUnknown* for postal codes.

polarity

```
ccIDDefs::Polarity polarity() const;
void polarity(ccIDDefs::Polarity polarity);
```

- `ccIDDefs::Polarity polarity() const;`

Returns the polarity of symbols for which the tool will search. The returned value is one of the following:

ccIDDefs::eDarkOnLight
ccIDDefs::eLightOnDark
ccIDDefs::ePolarityUnknown

If **polarity()** is *ccIDDefs::ePolarityUnknown*, the tool will search for both dark-on-light and light-on-dark symbols.

- `void polarity(ccIDDefs::Polarity polarity);`

Sets the polarity of symbols for which the tool will search. A value of ccIDDefs::eDarkOnLight specifies dark code elements on a light background; ccIDDefs::eLightOnDark specifies light code elements on a dark background.

If you specify ccIDDefs::ePolarityUnknown, the tool will search for symbols of either polarity.

Parameters

polarity The polarity to set. *polarity* must be one of the following values:

ccIDDefs::eDarkOnLight
ccIDDefs::eLightOnDark
ccIDDefs::ePolarityUnknown

Throws

ccIDDefs::BadParams
polarity is invalid.

Notes

Symbol polarity only applies to postal and PDF417 codes; it is ignored for linear symbologies.

isPostalOmniDirectional

```
bool isPostalOmniDirectional() const;

void isPostalOmniDirectional(
   bool isPostalOmniDirectional);
```

- `bool isPostalOmniDirectional() const;`

  Returns whether or not the tool searches for postal codes at arbitrary orientations. If the value is false, the tool only finds codes that are approximately aligned to the X-axis of the image. If the value is true, the tool finds codes at any orientation.

- `void isPostalOmniDirectional(`
  `bool isPostalOmniDirectional);`

  Sets whether or not the tool searches for postal codes at arbitrary orientations. If the value is false, the tool only finds codes that are approximately aligned to the X-axis of the image. If the value is true, the tool finds codes at any orientation.

  Specifying omnidirectional search may cause the tool to run more slowly.

  Parameters

  isPostalOmniDirectional

  specify true to enable omnidirectional search, false to search only for horizontally aligned codes.

  Notes

  This parameter is ignored for non-postal symbologies.

operatingMode

```
ccIDDefs::OperatingMode operatingMode() const;

void operatingMode(ccIDDefs::OperatingMode operatingMode);
```

- `ccIDDefs::OperatingMode operatingMode() const;`

  Gets the tool's operating mode.

- `void operatingMode(ccIDDefs::OperatingMode operatingMode);`

Sets the tool's operating mode.

The default value is *ccIDDefs::kDefaultOperatingMode*

Notes

The operating mode applies to all bar code types.

The e1DMax mode is recommended for dealing with hard-to-read codes. You must use the setter to explicitly specify *e1DMax* in order to activate that mode because the default mode is *eStandard*.

Throws

ccIDDefs::BadParams
The *operatingMode* argument is invalid.

Operators

operator== `bool operator== (const ccIDSearchParams& that) const;`

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDSearchParams

ccIDSearchResult

```
#include <ch_cvl/id.h>

class ccIDSearchResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the search result for an individual symbol. A search result is generated for each symbol found during a run of the ID tool.

Constructors/Destructors

ccIDSearchResult

```
ccIDSearchResult();
```

Constructs an object with the following default values:

location: (0, 0)
angle: 0
moduleSize: -1.0
mirror: *ccIDDefs::eMirroredUnknown*
polarity: *ccIDDefs::ePolarityUnknown*
has2DInfo: false

Public Member Functions

location

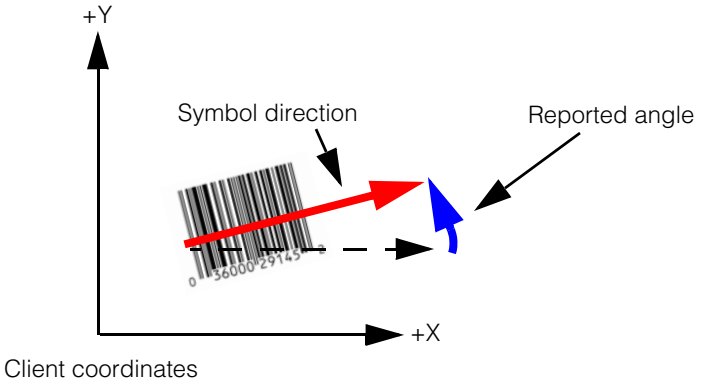
```
cc2Vect location() const;
```

Returns the center position of the found symbol in client coordinates.

angle

```
ccRadian angle() const;
```

Returns the found angle of the symbol, in radians. The returned value is the angle between the positive X-axis in client coordinate space and the symbol angle, which is the positive reading direction. The symbol direction is perpendicular to the code modules in the case of 1D codes.



The returned value is in the range of 0 through 2**TT.**

Notes

If the symbol is distorted, angle is the average value across the symbol.

moduleSize

```
double moduleSize() const;
```

Returns the size of a module of the symbology in client units. If the module size was not measured, this function returns -1.

mirror

```
ccIDDefs::Mirrored mirror() const;
```

Returns whether or not the found symbol was mirrored. This function returns one of the following values:

ccIDDefs::eMirroredFalse
ccIDDefs::eMirroredTrue
ccIDDefs::eMirroredUnknown

Notes

A value of *ccIDDefs::eMirroredUnknown* indicates the condition was not determined.

polarity

```
ccIDDefs::Polarity polarity() const;
```

Returns the polarity of the found symbol. This function returns one of the following values:

ccIDDefs::eDarkOnLight
ccIDDefs::eLightOnDark
ccIDDefs::ePolarityUnknown

Notes

A value of *ccIDDefs::ePolarityUnknown* indicates the polarity was not determined.

has2DInfo

```
bool has2DInfo() const;
```

Indicates whether 2D information is available for the found symbol. This function returns false for all linear, postal, and stacked codes.

numRows2D

```
c_Int32 numRows2D() const;
```

Returns the number of rows in the 2D symbol.

Throws

ccIDDefs::NotComputed
There is no 2D information (**has2DInfo()** returns false).

numCols2D

```
c_Int32 numCols2D() const;
```

Returns the number of columns in the 2D symbol.

Throws

ccIDDefs::NotComputed
There is no 2D information (**has2DInfo()** returns false).

symbolRegion `const ccPolyline& symbolRegion() const;`

Returns a polyline that describes the boundary of the symbol's found region.

The polyline traverses the four corners of the symbol, where the corners are defined by the ANSI or AIM specification for the symbol type:

Corner 0: upper left
Corner 1: upper right
Corner 2: lower right
Corner 3: lower left

For mirrored symbols, the corners are mirrored.

Throws

ccIDDefs::NotComputed
This **ccIDSearchResult** was default constructed.

Operators

operator== `bool operator== (const ccIDSearchResult& that) const;`

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDSubResult

```
#include <ch_cvl/id.h>

class ccIDSubResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing all of the result information for a subresult. For composite symbols, each **ccIDResult** contains multiple **ccIDSubResult**s. For non-composite symbols, only a single **ccIDSubResult** is present.

Constructors/Destructors

ccIDSubResult

```
ccIDSubResult();
```

Constructs this object with the following values:

isFound: false
isDecoded: false
failureCode: *ccIDDefs::eFailureNotApplicable*

Public Member Functions

isFound

```
bool isFound() const;
```

Returns true if a symbol was found, false otherwise.

In order for a symbol to be found, its symbology must be enabled in the **ccIDSearchParams** object supplied to the tool.

If the function returns false, additional information about the failure may be obtained by calling **failureCode()**.

isDecoded

```
bool isDecoded() const;
```

Returns true if the symbol was both found and decoded successfully, false otherwise.

ccIDSubResult

failureCode `ccIDDefs::FailureCode failureCode() const;`

If either **isFound()** or **isDecoded()** returns false, this function returns a failure code that describes the reason for the search or decode failure.

The returned value is a member of the **ccIDDefs::FailureCode** enumeration. If no failure code is available (either because this object is default-constructed or the search and decode succeeded), the function returns *ccIDDefs::eFailureNotApplicable*.

Notes

The codes returned by this function are for diagnostic and informational purposes only. The returned codes may change in future releases.

searchResult `const ccIDSearchResult& searchResult() const;`

Returns the search result from this subresult.

Throws

ccIDDefs::NotComputed
There are no results (**isFound()** is false).

decodeResult `const ccIDDecodeResult& decodeResult()const;`

Returns the decode result from this subresult.

Throws

ccIDDefs::NotComputed
There is no decode result(**isDecoded()** is false).

qualityResult

`const ccIDQualityResult& qualityResult() const;`

Returns the quality result from this subresult.

Notes

When the symbol was decoded but **ccIDDecodeParams::qualityOption** was set to *ccIDQualityDefs::eQualityOptionNone*, a default constructed **ccIDQualityResult** is returned.

Throws

ccIDDefs::NotComputed
There is no decode result(**isDecoded()** is false).

draw `void draw(ccGraphicList& glist, c_UInt32 drawMode) const;`

Appends graphics for this subresult to the supplied graphics list. *drawMode* is used to select which graphics are drawn and must be a bitwise combination of the following values:

ccIDDefs::eDrawCenter
ccIDDefs::eDrawBoundary

ccIDDefs::eDrawCenter draws a cross at this subresult's location; *ccIDDefs::eDrawBoundary* draws the boundary of this subresult, which is a polygon connecting the corners.

Graphics are drawn in *ccColor::green* if the subresult was successfully decoded, *ccColor::red* if the subresult was found but not decoded, and no graphics are drawn if **isFound()** is false.

Parameters

glist	The graphics list to which to append the graphic.
drawMode	The drawing mode.

Notes

All graphics are drawn in client coordinates.

Operators

operator== `bool operator== (const ccIDSubResult& that) const;`

Returns true if the supplied object is equal to this one.

Parameters

that	The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccIDSubResult

ccImageFont

```
#include <ch_cvl/synfont.h>

class ccImageFont;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccImageFont** class represents a set of characters that are each specified as bitmaps rather than as some synthetic specification.

Constructors/Destructors

The default constructor, copy constructor, assignment operator, and destructor are used.

Public Member Functions

name

```
const ccCvlString& name() const;

void name(const ccCvlString& n);
```

- `const ccCvlString& name() const;`

  Gets the name of this font.

- `void name(const ccCvlString& n);`

  Sets the name of this font.

  Parameters

n	The name of this font

characters

```
const cmStd vector<ccImageFontChar>& characters() const;

void characters(const cmStd vector<ccImageFontChar>& c,
  bool makeUniqueInstanceKeys = false);
```

- `const cmStd vector<ccImageFontChar>& characters() const;`

  Gets the characters in this font.

- `void characters(const cmStd vector<ccImageFontChar>& c, bool makeUniqueInstanceKeys = false);`

  Sets the characters in this font.

  Parameters

  c The characters in this font.

  makeUniqueInstanceKeys
  If true, then characters will all be assigned unique keys; specifically, later characters in the set will be reassigned unique instance values.

  Throws

  ccSynFontDefs::SameKey
  Two or more characters have the same key and *makeUniqueInstanceKeys* is false.

hasCharacter

```
bool hasCharacter(c_Int32 key) const;
```

Returns true if a character with the specified key exists in this font.

Parameters

key The specified key.

character

```
const ccImageFontChar& character(c_Int32 key) const;
```

Returns the character with the specified key.

Parameters

key The specified key.

Throws

ccSynFontDefs::BadParams
No character has the specified key.

add

```
void add(const ccImageFontChar& imageChar);
```

Adds the character to the font.

Parameters

imageChar The character to be added.

Throws

ccSynFontDefs::SameKey
Two or more characters have the same key.

---

remove

```
void remove(c_Int32 key);
void remove(const cmStd vector<c_Int32>& keys);
```

- `void remove(c_Int32 key);`

Removes the character with the specified key from the font.

Parameters

key The specified key.

Throws

ccSynFontDefs::BadParams
No character has the specified key.

- `void remove(const cmStd vector<c_Int32>& keys);`

Removes the characters with the specified keys from the font.

Parameters

keys The specified keys.

Throws

ccSynFontDefs::BadParams
No character has each specified key.

removeAll

```
void removeAll();
```

Removes all characters from the font.

leading

```
cc2Vect leading() const;
void leading(const cc2Vect&);
```

- `cc2Vect leading() const;`

  Gets the leading for this font.

- `void leading(const cc2Vect& leadingForFont);`

  Sets the leading for this font.

  Parameters

  leadingForFont The new leading for this font.

encloseCellRect

```
ccRect encloseCellRect() const;
```

Gets the enclosing cell rect for the font, which encloses all of the individual character cell rects.

encloseMarkRect

```
ccRect encloseMarkRect() const;
```

Gets the enclosing mark rect for the font, which encloses all of the individual character mark rects.

assignAlphabet

```
void assignAlphabet(const ccOCAlphabet& alphabet,
  bool makeDuplicateCharsWithUnicodeKeys = true);
```

Resets this font to be essentially identical to the OCV alphabet.

Parameters

alphabet The OCV alphabet.

makeDuplicateCharsWithUnicodeKeys
If true, duplicate characters are made with unicode keys.

Notes

Since an OCV alphabet does not specify all the information that needs to be specified for an image font, reasonable values will be chosen:

1) The advance for each character will be set equal to its width.

2) The mark rect for non-blank characters will be set equal to the cell rect.

3) The leading for the font will be set equal to the height of the tallest character.

If *makeDuplicateCharsWithUnicodeKeys* is true, then in addition to creating characters with the keys as they already are in the alphabet, copies of those characters will be made with key values that are typical ASCII values, on the assumption that the Unicode key value is in the upper 16 bits of the alphabet keys; such an encoding scheme is used by the VisionPro OCV font editor when it saves font files in CVL (".ocf") format.

makeAlphabet

```
ccOCAlphabetPtrh makeAlphabet() const;
```

Creates an OCV alphabet that is identical to this font.

moveCharOrigins

```
void moveCharOrigins( ccImageFontChar::Origin origin =
  ccImageFontChar::eDetectedMarkLowerLeft);
```

Move the origins of all characters as specified.

Parameters

origin The origin.

Notes

For blanks, *eDetectedMarkLowerLeft* and *eSpecifiedMarkLowerLeft* will compute a "reasonable" value based on the origin locations of other characters.

detectCharMarkRects

```
void detectCharMarkRects();
```

Detects the mark rects of all the characters.

detectCharPolarities

```
void detectCharPolarities(bool onlyIfUnknown = true,
  bool expectAllSamePolarity = true);
```

Detects the polarities of all the characters.

Parameters

onlyIfUnknown Only if the polarity is unknown.

expectAllSamePolarity
Expects that the polarity information is the same for all characters.

leftJustifyChars

```
void leftJustifyChars(double leftPadding = 2.);
```

Left-justifies all the characters.

Parameters

leftPadding The left padding value.

save

```
void save(const ccCvlString& filename,
  bool saveAsOcf = false) const;
```

Saves the font to a file with the given filename. If *saveAsOcf* is true, the file will be saved in the older ".ocf" file format, which loses some information in the metrics.

Parameters

filename The name of the file.

saveAsOcf If true, the file will be saved in the older ".ocf" file format.

load

```
void load(const ccCvlString& filename);
```

Loads the font from the file with the given filename; the file may be in either ".ocf" or ".ocm" format. The font will be unchanged if the load fails.

Parameters

filename The name of the file.

Throws

ccSynFontDefs::BadFileFormat if the file does not contain an image font; it can also throw other exceptions (for example, *ccArchive::Eof, ccArchive::UnknownVersion,* and so on).

tryLoad

```
bool tryLoad(const ccCvlString& filename);
```

Tries to load the font from the file with the given filename; the file may be in either ".ocf" or ".ocm" format. Returns true if successful or false if the file was in a bad format.

Caution *Does not throw* ccSynFontDefs::BadFileFormat *if the file is in a bad format.*

The font will be unchanged if the load fails.

Parameters

filename The name of the file.

makeUniqueInstanceKey

```
c_Int32 makeUniqueInstanceKey(c_Int32 key) const;
```

Returns a unique key to be used for a new instance of the given key, with the same Unicode character value.

Parameters

key The given key.

ccImageFont

ccImageFontChar

```
#include <ch_cvl/synfont.h>

class ccImageFontChar;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccImageFontChar** class represents a character from an image font, that is, a font that consists of character bitmaps rather than as some synthetic specification.

Constructors/Destructors

ccImageFontChar

```
ccImageFontChar();
```

Constructs a blank character with a key of -1 and polarity of *eUnknown*.

Notes

The default copy constructor, assign operator, and destructor are used.

Enumerations

Origin

```
enum Origin;
```

Type of origin for a character.

Value	Meaning
eImageCenter = 0	Center of character image
eImageLowerLeft	Lower left of character image
eSpecifiedMarkLowerLeft	Lower left of mark rect
eDetectedMarkLowerLeft	Lower left of detected mark rect

Public Member Functions

key

```
c_Int32 key() const;
void key(c_Int32 key);
```

- `c_Int32 key() const;`

  Gets character key.

- `void key(c_Int32 key);`

  Sets character key.

  Parameters

  key The new character key.

name

```
const ccCvlString& name() const;
void name(const ccCvlString& n);
```

- `const ccCvlString& name() const;`

  Gets the name for this character.

- `void name(const ccCvlString& n);`

  Sets the name for this character.

  Parameters

  n The name for this character.

description

```
const ccCvlString& description() const;
void description(const ccCvlString& n);
```

- `const ccCvlString& description() const;`

  Gets the description for this character.

- `void description(const ccCvlString& n);`

  Sets the description for this character.

Parameters

n The description.

polarity

```
ccSynFontDefs::Polarity polarity() const;

void polarity(ccSynFontDefs::Polarity p);
```

- `ccSynFontDefs::Polarity polarity() const;`

  Gets the polarity for this character.

- `void polarity(ccSynFontDefs::Polarity p);`

  Sets the polarity for this character.

  Parameters

  p The polarity for this character.

  Throws

  ccSynFontDefs::BadParams
  The polarity is not valid.

image

```
const ccPelBuffer_const<c_UInt8>& image() const;

void image(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelRect& maxRootCopyRect = ccPelRect());
```

- `const ccPelBuffer_const<c_UInt8>& image() const;`

  Gets the image for this character.

  Throws

  ccSynFontDefs::BadParams
  maxRootCopyRect is not a null rect and does not contain the image's window.

- `void image(const ccPelBuffer_const<c_UInt8>& image, const ccPelRect& maxRootCopyRect = ccPelRect());`

  Sets the image for this character.

  A deep copy of the image is made, including pixels from the root outside the image if specified.
  maxRootCopyRect may be a null rect to indicate that only the image's window should be copied; otherwise, it should enclose the image's window.

Parameters

image The image for this character.

Throws

ccSynFontDefs::BadParams

maxRootCopyRect is not a null rect and does not contain the image's window.

---

mask

```
const ccPelBuffer_const<c_UInt8>& mask() const;

void mask(const ccPelBuffer_const<c_UInt8>& mask,
  const ccPelRect& maxRootCopyRect = ccPelRect());
```

- `const ccPelBuffer_const<c_UInt8>& mask() const;`

Gets the mask for this character.

Notes

Getter returns an unbound image buffer if no mask has been specified for this character.

Throws

ccSynFontDefs::BadParams

maxRootCopyRect is not a null rect and does not contain the image's window.

- `void mask(const ccPelBuffer_const<c_UInt8>& mask, const ccPelRect& maxRootCopyRect = ccPelRect());`

Sets the mask for this character.

A deep copy of the image is made, including pixels from the root outside the image if specified.
maxRootCopyRect may be a null rect to indicate that only the image's window should be copied; otherwise, it should enclose the image's window.

Parameters

mask The mask for this character.

Throws

ccSynFontDefs::BadParams

maxRootCopyRect is not a null rect and does not contain the image's window.

imageArea

```
ccPelRect imageArea() const;
void imageArea(const ccPelRect& area);
```

- `ccPelRect imageArea() const;`

  Gets the character area for this character in image coordinates.

- `void imageArea(const ccPelRect& area);`

  Sets the character area for this character in image coordinates.

  Parameters

  area The character area for this character.

moveOrigin

```
bool moveOrigin(Origin origin = eDetectedMarkLowerLeft);
void moveOrigin(const cc2Vect& origin);
```

- `bool moveOrigin(Origin origin = eDetectedMarkLowerLeft);`

  Moves the origin to the specified location. Returns true if the origin could be moved as specified.

  Parameters

  origin The specified location to become the new origin.

  Notes

  For blanks, *eDetectedMarkLowerLeft* and *eSpecifiedMarkLowerLeft* are treated as *eImageLowerLeft*.

- `void moveOrigin(const cc2Vect& origin);`

  Moves the origin to the specified location, that is, makes the specified point become (0, 0) in client coordinates.

  Parameters

  origin The specified location to become the new origin.

metrics

```
const ccFontCharMetrics& metrics() const;
void metrics(const ccFontCharMetrics& m);
```

- `const ccFontCharMetrics& metrics() const;`

  Gets the metrics for this character.

- `void metrics(const ccFontCharMetrics& m);`

  Sets the metrics for this character.

  Parameters

  m The metrics for this character.

detectMarkRect

```
bool detectMarkRect();
```

Computes the detected mark rect and store it in the metrics.

Returns whether the mark rect detection was successful.

Notes

Detection will never be successful for a blank.

detectPolarity

```
bool detectPolarity(bool onlyIfPolarityUnknown = true);
```

Attempts to detect the polarity. Returns whether the polarity detection was successful.

Notes

The detection will not be successful for a blank whose polarity is currently unknown.

Parameters

onlyIfPolarityUnknown
Only if the polarity is unknown.

trim

```
void trim(const ccRect& rect, bool padIfLarger = true,
  bool resetAdvance = true,
  bool pelsOutsideImageAreaAreBackground = false);
```

Trims this character to be at most the greatest common rectangle of the character with the specified client rectangle. This may affect the cell rect, mark rect, *detectedMarkRect*, and *imageArea*.

Notes

The character will not be expanded if the specified rect lies partially outside the current character.

If *resetAdvance* is true, the advance will be set equal to the width of the new cell rect.

Parameters

rect The specified client rectangle.

Notes

This function may be useful when trying to get a set of characters to have a common size.

Throws

ccSynFontDefs::BadParams
The rect would cause the character to become empty.

trimRelative

```
void trimRelative(double left,
 double right,
 double top,
 double bottom,
 bool padIfLarger = true,
 bool resetAdvance = true,
 bool pelsOutsideImageAreaAreBackground = false);
```

Trims this character using its current *cellRect* modified by the specified amounts; negative values mean that the indicated border should be grown rather than shrunk. This may affect the cell rect, mark rect, *detectedMarkRect*, and *imageArea*.

Notes

The character will not be expanded if the specified rect lies partially outside the current character.

Parameters

left The amount by which to shrink (positive values) or grow (negative values) the left edge of the rectangle.

right The amount by which to shrink (positive values) or grow (negative values) the right edge of the rectangle.

top The amount by which to shrink (positive values) or grow (negative values) the top edge of the rectangle.

bottom The amount by which to shrink (positive values) or grow (negative values) the bottom edge of the rectangle.

Notes

This function may be useful when trying to get a set of characters to have a common size.

Throws

ccSynFontDefs::BadParams
The rect would cause the character to become empty.

trimY

```
void trimY(double minY,
  double maxY,
  bool padIfLarger = true,
  bool pelsOutsideImageAreaAreBackground = false);
```

Trims this character to be at most the greatest common rectangle of the character within the specified client lines. This may affect the cell rect, mark rect, *detectedMarkRect*, and *imageArea*.

Notes

The character will not be expanded if the specified rect lies partially outside the current character.

This function may be useful when trying to get a set of characters to have a common size.

Parameters

minY The y-position of the starting client line.

maxY The y-position of the ending client line.

Throws

ccSynFontDefs::BadParams
The rect would cause the character to become empty.

leftJustify

```
void leftJustify(double leftPadding = 2.);
```

Shifts this character so that the specified amount of background (in client units) exists to the left of the detected mark, without changing the character's width.

Parameters

leftPadding The left padding value.

ccImageFontChar

ccImageRegisterParams

```
#include <ch_cvl/imgreg.h>

class ccImageRegisterParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters for the Image Registration tool.

Constructors/Destructors

ccImageRegisterParams

```
ccImageRegisterParams();

ccImageRegisterParams(Method method, Algorithm algo);

ccImageRegisterParams(Method method, Algorithm algo,
  const ccIPair & start);
```

- `ccImageRegisterParams()`

  Constructs a **ccImageRegisterParams** with the default algorithm and scoring method and with no starting point.

- `ccImageRegisterParams(Method method, Algorithm algo);`

  Constructs a **ccImageRegisterParams** using the specified method and algorithm. No starting point is set.

  Parameters

  method — The scoring method to use. *method* must be one of the following values:

ccImageRegisterParams::eCorrelation
ccImageRegisterParams::eSumAbsDifferences
ccImageRegisterParams::kDefaultMethod

algo — The scoring algorithm to use. *algo* must be one of the following values:

ccImageRegisterParams::eLocalSearch
ccImageRegisterParams::eExhaustiveSearch
ccImageRegisterParams::kDefaultAlgorithm

- `ccImageRegisterParams(Method method, Algorithm algo, const ccIPair & start)`

  Constructs a **ccImageRegisterParams** using the specified method, algorithm, and starting point.

  Parameters

  method — The scoring method to use. *method* must be one of the following values:

  ccImageRegisterParams::eCorrelation
  ccImageRegisterParams::eSumAbsDifferences
  ccImageRegisterParams::kDefaultMethod

  algo — The scoring algorithm to use. *algo* must be one of the following values:

  ccImageRegisterParams::eLocalSearch
  ccImageRegisterParams::eExhaustiveSearch
  ccImageRegisterParams::kDefaultAlgorithm

  start — The whole-pixel starting point, specified in the source image's image coordinate system.

Enumerations

Algorithm

```
enum Algorithm;
```

This enumeration defines the Image Registration algorithms.

Value	Meaning
eLocalSearch	Search for the local minimum or maximum score.
eExhaustiveSearch	Exhaustively search the entire source image for the global minimum or maximum score.
kDefaultAlgorithm	The default algorithm defined in *imgreg.h*

Method

```
enum Method;
```

This enumeration defines the Image Registration tool scoring methods.

Value	Meaning
eCorrelation	Normalized correlation coefficient is computed between pixels in reference image and corresponding pixels in source image. Scores range from -1.0 (perfect mismatch) to +1.0 (perfect match).
eSumAbsDifferences	Sum of absolute differences between pixels in reference image and corresponding pixels in source image. Scores range from 0.0 (perfect match) to a value equal to 256 times the number of pixels in the reference image. Larger values indicate poorer matches.

Value	Meaning
eHighAccuracyCorrelation	The normalized correlation coefficient is computed, as described for *eCorrelation*, but an advanced technique is used to more accurately determine the sub-pixel location at which the score is the highest.
kDefaultMethod	The default scoring method defined in *imgreg.h*

Public Member Functions

algorithm

```
void algorithm(Algorithm algo);
Algorithm algorithm() const;
```

- `void algorithm(Algorithm algo);`

  Sets the algorithm used by this **ccImageRegisterParams**.

  Parameters

  algo — The scoring algorithm to use. *algo* must be one of the following values:

  ccImageRegisterParams::eLocalSearch
  ccImageRegisterParams::eExhaustiveSearch
  ccImageRegisterParams::kDefaultAlgorithm

- `Algorithm algorithm() const;`

  Returns the algorithm used by this **ccImageRegisterParams**. This function returns one of the following values:

  ccImageRegisterParams::eLocalSearch
  ccImageRegisterParams::eExhaustiveSearch

method

```
void method(Method method);
Method method() const;
```

- `void method(Method method);`

  Sets the scoring method used by this **ccImageRegisterParams**.

Parameters

method — The scoring method to use. *method* must be one of the following values:

ccImageRegisterParams::eCorrelation
ccImageRegisterParams::eSumAbsDifferences
ccImageRegisterParams::eHighAccuracyCorrelation
ccImageRegisterParams::kDefaultMethod

- `Method method() const;`

Returns the scoring method used by this **ccImageRegisterParams**. This function returns one of the following values:

ccImageRegisterParams::eCorrelation
ccImageRegisterParams::eSumAbsDifferences
ccImageRegisterParams::eHighAccuracyCorrelation

startPoint

```
void startPoint(const ccIPair &start);

ccIPair startPoint() const;
```

- `void startPoint(const ccIPair &start);`

Sets the starting point for the search. *start* is specified in the image coordinate system of the source image and should be within a few pixels of the upper-left corner of the part of the source image that corresponds to the reference image.

Parameters

start — The starting search point, in the source image's image coordinate system.

- `ccIPair startPoint() const;`

Returns the starting point for the search. The point is in the source image's image coordinate system.

isStartPointSet

```
bool isStartPointSet() const;
```

Returns true if this **ccImageRegisterParams**'s starting point has been set, either at construction or by calling **startPoint()**. Returns false if the starting point has not been set.

ccImageRegisterParams

ccImageRegisterResults

```
#include <ch_cvl/imgreg.h>

template <class P>class ccImageRegisterResults;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single result produced by the Image Registration tool.

Constructors/Destructors

ccImageRegisterResults

```
ccImageRegisterResults();
```

Constructs a **ccImageRegisterResults** which you can supply to **cfImageRegister()**.

Enumerations

EdgeHit

```
enum EdgeHit;
```

This enumeration defines the possible sides upon which the reference image might be clipped by the source image.

Value	Meaning
eLeftEdge	The left side of the reference image is clipped by the source image.
eTopEdge	The top of the reference image is clipped by the source image.
eRightEdge	The right side of the reference image is clipped by the source image.
eBottomEdge	The bottom of the reference image is clipped by the source image.

Public Member Functions

position

```
cc2Vect position() const;
```

Returns the location in the source image that corresponds to the upper-left corner of the reference image. The position is returned in the source image's image coordinate system.

score

```
double score() const;
```

Returns the score of this results.

If the sum of absolute differences scoring method is used, then scores range from 0 to 256 times the number of pixels in the reference image, with a score of 0.0 indicating a perfect match and larger scores indicating poorer matches.

If the normalized correlation scoring method is used, then scores range from -1.0, which indicates a perfect mismatch, to +1.0, which indicates a perfect match.

edgeHit

```
c_UInt32 edgeHit() const;
```

Returns which, if any, of the sides of the reference image were clipped by the source image. If the reference image was clipped then the accuracy of the result can be reduced.

This function returns 0 to indicate that no clipping occurred. If clipping occurred, then this function returns a value formed by ORing together 1 or more of the following values:

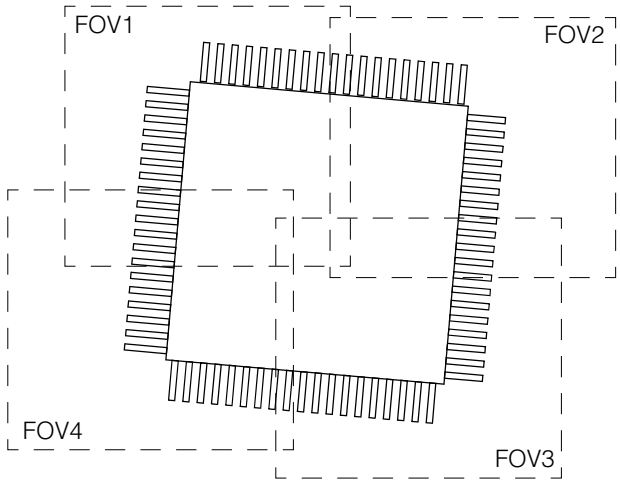
eLeftEdge
eTopEdge
eRightEdge
eBottomEdge

ccImageStitch

```
#include <stitch.h>

class ccImageStitch;
```

This class encapsulates the Image Stitching tool used to stitch multiple source images together into a single destination image. In applications where a single image field-of-view (FOV) cannot capture the entire scene, you can capture multiple FOVs and stitch them together to make a single result image that can then be processed by other vision tools. For example, the large leaded SMD device shown below is captured in four FOVs. Using the Image Stitching tool you can stitch these four images together to make a single result image of the entire device.



The FOV images can be provided by a single moving camera, or by multiple fixed cameras. Cameras need not be the same type or size but all FOVs must be calibrated so that the client coordinate transform associated with each image targets a common client coordinate space.

The following steps outline how you might typically use the Image Stitching tool in an application.

1. Construct a **ccImageStitch** object.
2. Call **init()** to initialize the tool and specify the source and destination images.
3. Call **clearImages()** to prepare the stitching tool.
4. Acquire an image to stitch.

5. Call **addImage()** to stitch the acquired image.

   Loop back to step 4 until all images are processed.

6. Call **stitchedImage()** to get the stitched result image, and **stitchedMaskImage()** to get the result image mask.

7. Run vision tools on the stitched image result or use the stitched image in some other way.

8. Return to Step 3 for the next stitching operation.

Constructors/Destructors

ccImageStitch

```
ccImageStitch();
```

Constructs an Image Stitch object with empty data buffers.

Notes

The default copy and assign operators make shallow copies.

Public Member Functions

reset

```
void reset();
```

Releases all resources and returns this object to the default constructed state.

computeEnclosingStitchedRect

```
ccPelRect computeEnclosingStitchedRect(
  const cmStd vector<ccPelRect>& srcImageWindows,
  const cmStd vector<cc2Xform>& srcImageFromClientXforms,
  const cc2Xform& dstImageFromClientXform) const;
```

Computes a destination image window that encloses all the specified source image windows after they are properly stitched together in the common client space. The returned destination image window maps to an affine rectangle region in the client space through the inverse of *dstImageFromClientXform*.

This is an optional utility function you can use to compute a bounding box result image window with an orientation you specify in *dstImageFromClientXform*. Use the **ccPelRect** returned by this function as *dstImageWindow* and the *dstImageFromClientXform* specified here when calling the second **init()** overload.

Parameters

srcImageWindows

A vector of source windows containing all of the images you wish to stitch together.

srcImageFromClientXforms
A vector of transforms corresponding to the windows in *srcImageWindows* above. These are the calibration transforms that map source image pixels into the common client space of the result.

dstImageFromClientXform
The transform that maps the result client space into the destination image.

Throws

ccImageStitchDefs::BadInput
if any one of the following is true:
- the sizes of the two input vectors differ,
- the two input vectors are empty,
- any element of *srcImageWindows* is null

ccImageStitchDefs::Singular
if any one of the following is true:
- any element of *srcImageFromClientXforms* is singular,
- *dstImageFromClientXform* is singular

Notes

This function can be called regardless of whether **isInitialized()** returns true or false.

This function does not alter the state of the object (const member function). In particular, it does not affect the value returned by **isInitialized()**.

The window of the image returned by **stitchedImage()** will be decided by **init()**, and is not affected by this function.

The elements in the *srcImageWindows* and *srcImageFromClientXforms* vectors must correspond to each other.

isInitialized

```
bool isInitialized() const;
```

Returns whether **init()** has been called successfully. The default is *false*

init

```
void init(
  const cmStd vector<ccPelRect>& srcImageWindows,
  const cmStd vector<cc2Xform>& srcImageFromClientXforms,
```

```
  ccImageStitchDefs::OutputMode
                        outputMode=ccImageStitchDefs::eBlend,
  c_Int32 valueForEmptyRegion = 0);

void init(
  const cmStd vector<ccPelRect>& srcImageWindows,
  const cmStd vector<cc2Xform>& srcImageFromClientXforms,
  const ccPelRect& dstImageWindow,
  const cc2Xform& dstImageFromClientXform,
  ccImageStitchDefs::OutputMode
                        outputMode=ccImageStitchDefs::eBlend,
  c_Int32 valueForEmptyRegion = 0);
```

Initializes internal data buffers and calls **clearImages()** to prepare the tool for image stitching.

-
```
void init(
  const cmStd vector<ccPelRect>& srcImageWindows,
  const cmStd vector<cc2Xform>& srcImageFromClientXforms,
  ccImageStitchDefs::OutputMode
                        outputMode=ccImageStitchDefs::eBlend,
  c_Int32 valueForEmptyRegion = 0);
```

Initializes the tool and sets the internal data buffers so that the result image window covers a region in client space that encloses all the specified source image windows. The tool is set to use the first element of *srcImageFromClientXforms* for *dstImageFromClientXform*.

Parameters

srcImageWindows

A vector of source windows containing the windows of all of the images you wish to stitch together.

srcImageFromClientXforms

A vector of transforms corresponding to the windows in *srcImageWindows* above. These are the calibration transforms that map source image pixels into the common client space of the result.

outputMode Specifies how pixel values from source images are combined to form a result pixel value in the destination image. Must be either *eBlend* or *eOverwrite*. In *eOverwrite* mode, each result pixel value is the value from the most recently stitched input image with a corresponding pixel. Earlier pixel values are overwritten. In *eBlend* mode each result pixel value is a blend of all of the corresponding pixel values in all the input images.

valueForEmptyRegion
Specifies the 8-bit pixel value assigned to destination image pixels that had no corresponding pixels in any input image.

Throws

ccImageStitchDefs::BadInput
if any one of the following is true:
- the sizes of the two input vectors differ,
- the two input vectors are empty,
- any element of *srcImageWindows* is null,
- *dstImageWindow* is null,
- *outputMode* is not a valid value for *ccImageStitchDefs::OutputMode*

ccImageStitchDefs::Singular
if *srcImageFromClientXforms* is singular

Notes

The elements in the two input vectors must correspond to each other.

Only the low-order 8 bits of *valueForEmptyRegion* are used.

- ```
  void init(
    const cmStd vector<ccPelRect>& srcImageWindows,
    const cmStd vector<cc2Xform>& srcImageFromClientXforms,
    const ccPelRect& dstImageWindow,
    const cc2Xform& dstImageFromClientXform,
    ccImageStitchDefs::OutputMode
                          outputMode=ccImageStitchDefs::eBlend,
    c_Int32 valueForEmptyRegion = 0);
  ```

Initializes the tool and sets the internal data buffers so that the result image window is *dstImageWindow*, using *dstImageFromClientXform* to map the result from client space.

Parameters

srcImageWindows
A vector of source windows containing the windows of all of the images you wish to stitch together.

srcImageFromClientXforms
A vector of transforms corresponding to the windows in *srcImageWindows* above. These are the calibration transforms that map source image pixels into the common client space of the result.

dstImageWindow
The destination image window.
  ```

dstImageFromClientXform
The transform that maps the result client space into the destination image.

outputMode Specifies how pixel values from source images are combined to form a result pixel value in the destination image. Must be either *eBlend* or *eOverwrite*. In *eOverwrite* mode, each result pixel value is the value from the most recently stitched input image with a corresponding pixel. Earlier pixel values are overwritten. In *eBlend* mode each result pixel value is a blend of all of the corresponding pixel values in all the input images.

valueForEmptyRegion
Specifies the 8-bit pixel value assigned to destination image pixels that had no corresponding pixels in any input image.

Throws

ccImageStitchDefs::BadInput
if any one of the following is true:
- the sizes of the two input vectors differ,
- the two input vectors are empty,
- any element of *srcImageWindows* is null,
- *dstImageWindow* is null,
- *outputMode* is not a valid value for *ccImageStitchDefs::OutputMode*

ccImageStitchDefs::Singular
if either of the following is true:
- any element of *srcImageFromClientXforms* is singular,
- *dstImageFromClientXform* is singular

Notes

The elements in the two vectors must correspond to each other.

Only the lowest 8 bits of *valueForEmptyRegion* are used.

outputMode `ccImageStitchDefs::OutputMode outputMode() const;`

Returns the output mode specified during initialization (see **init()**).

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

valueForEmptyRegion

```
c_Int32 valueForEmptyRegion() const;
```

Returns the pixel value specified during initialization for regions in the stitched image result not covered by any source image.

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

Notes

Although only the low-order 8 bits are used by the tool, the full 32 bits you specify are returned.

clearImages

```
void clearImages();
```

Clears the content of internal data buffers in preparation for a new stitching operation.

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

Notes

Sets **numImages()** to 0.

This function differs from **reset()** in that it just clears the contents of internal data buffers. It does not change image windows or client transforms and does not release any allocated resources.

addImage

```
void addImage(const ccPelBuffer_const<c_UInt8>& srcImage);

void addImage(const ccPelBuffer_const<c_UInt8>& srcImage,
              const ccPelBuffer_const<c_UInt8>& maskImage);
```

- ```
  void addImage(const ccPelBuffer_const<c_UInt8>& srcImage);
  ```

Adds an image for stitching.

Parameters

srcImage The image to add.
  ```

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

ccImageStitchDefs::BadImage
if *srcImage* is unbound or has a null window

ccImageStitchDefs::Singular
if any element of *srcImageFromClientXforms* is singular

ccImageStitchDefs::NotImplemented
if the *srcImage* client transform is non-linear,
or the output mode is *ccImageStitchDefs::eBlend* and the total number of images exceeds 255.

Notes

ccImageStitch does not support non-linear transforms. To stitch images with non-linear transforms such as the camera calibration transform, use the image warper (see **ccImageWarp**) to remove the non-linear distortion so that the images have linear client transforms.

If a time-out occurs (see **ccTimeout**) before this function completes, this function will automatically call **clearImages()** before throwing *ccTimeout::Expired*.

- ```
  void addImage(
    const ccPelBuffer_const<c_UInt8>& srcImage,
    const ccPelBuffer_const<c_UInt8>& maskImage);
  ```

Adds an image and an image mask for stitching. The *maskImage* window must be identical to that of *srcImage*. *maskImage* is aligned in image space to *srcImage*. The *maskImage* client transform is ignored.

maskImage is a binary mask image. Its pixel values must be either 0 or 255. A 0 specifies masked-out pixels (*srcImage* pixels to be ignored). A 255 identifies unmasked (effective) *srcImage* pixels.

Parameters

srcImage The image to add.

maskImage The *srcImage* mask.

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

ccImageStitchDefs::BadImage
if any one of the following is true:
- *srcImage* is unbound or has a null window,
  ```

- *maskImage* is unbound or has a null window,
- *maskImage*'s window is different from *srcImage*'s window,
- *maskImage* has pixel values other than 0 or 255

ccImageStitchDefs::Singular
if either of the following is true:
- any element of *srcImageFromClientXforms* is singular,
- *dstImageFromClientXform* is singular

ccImageStitchDefs::NotImplemented
if the *srcImage* client transform is non-linear,
or the output mode is *ccImageStitchDefs::eBlend* and the total number of images exceeds 255.

Notes

ccImageStitch does not support non-linear transforms. To stitch images with non-linear transforms such as the camera calibration transform, use the image warper (see **ccImageWarp**) to remove the non-linear distortion so that the images have linear client transforms.

If a time-out occurs (see **ccTimeout**) before this function completes, this function will automatically call **clearImages()** before throwing *ccTimeout::Expired*.

addImageWithWeights

```
void addImageWithWeights(
  const ccPelBuffer_const<c_UInt8>& srcImage,
  const ccPelBuffer<c_UInt8>& weightsImage);

void addImageWithWeights(
  const ccPelBuffer_const<c_UInt8>& srcImage,
  const ccPelBuffer<c_UInt8>& weightsImage,
  const ccPelBuffer_const<c_UInt8>& maskImage);
```

- ```
  void addImageWithWeights(
    const ccPelBuffer_const<c_UInt8>& srcImage,
    const ccPelBuffer<c_UInt8>& weightsImage);
  ```

Adds one image to the internal blending buffer using custom weights. This function may only be called in blending mode.

Parameters

srcImage The image to add.

weightsImage The *srcImage* weighing.
  ```

Throws

ccImageStitchDefs::NotInitialized
isInitialized() returns false.

ccImageStitchDefs::NotSupported
If called while in Overwrite mode.

ccImageStitchDefs::BadImage
Any of the following is true:
- *srcImage* is unbound or has a null window;
- *weightsImage* is unbound or has a null window;
- *weightsImage*'s window is different from *srcImage*'s window;

ccImageStitchDefs::Singular
srcImage's client transform is singular.

ccImageStitchDefs::NotImplemented
srcImage has a non-linear transform,
or the output mode is *ccImageStitchDefs::eBlend* and the total number of images exceeds 255.

- ```
  void addImageWithWeights(
    const ccPelBuffer_const<c_UInt8>& srcImage,
    const ccPelBuffer<c_UInt8>& weightsImage,
    const ccPelBuffer_const<c_UInt8>& maskImage);
  ```

Adds one image to the internal blending buffer using custom weights. This function may only be called in blending mode. *maskImage* is a binary mask image. Its pixel values must be either 0 or 255:

- 0 identifies masked-out pixels (to be ignored);
- 255 identifies unmasked (effective) pixels.

Parameters

srcImage The image to add.

weightsImage The *srcImage* weighing.

maskImage The *srcImage* mask.

Throws

ccImageStitchDefs::NotInitialized
isInitialized() returns false.

ccImageStitchDefs::NotSupported
If called while in Overwrite mode.
  ```

ccImageStitchDefs::BadImage
Any of the following is true:
- *srcImage* is unbound or has a null window;
- *weightsImage* is unbound or has a null window;
- *weightsImage*'s window is different from *srcImage*'s window;
- *maskImage* is unbound or has a null window;
- *maskImage*'s window is different from *srcImage*'s window;
- *maskImage* has pixel values other than 0 or 255.

ccImageStitchDefs::Singular
srcImage's client transform is singular.

ccImageStitchDefs::NotImplemented
srcImage has a non-linear transform,
or the output mode is *ccImageStitchDefs::eBlend* and the total number of images exceeds 255.

Notes

If you added one image where all the pixel weights were set to 2, and then a second where all the pixel weights were set to 1, then in the blended image buffer, each pixel value would equal 2/3 times the corresponding pixel in the first image plus 1/3 times that in the second image. However, if the weights corresponding to a region of pixels in the second image were set to zero, then, in the blended image, the corresponding pixels would equal the values of those in the first image.

When copying data from *srcImage* to internal data buffers, the correspondence is in the client space. In other words, for any image pixel position in *dstImage* (call it D), its corresponding position in the client space position (call it C) is found via *dstImage*'s client transform; then the corresponding image pixel position in *srcImage* (call it S) is found via *srcImage*'s client transform. The pixel value is read from S in *srcImage* and used to update (by weighted sum) the pixel value at D in *dstImage*.

weightsImage's window must be identical to that of *srcImage*. *weightsImage* is aligned in image space to *srcImage*. Its client transform is ignored.

Setting a pixel in the weights image to zero is identical to masking out that pixel, that is, setting a mask value of zero at that location.

maskImage's window must be identical to that of *srcImage*. *maskImage* is aligned in image space to *srcImage*. Its client transform is ignored.

If a time-out occurs (see **ccTimeout** in <*ch_cvl/attent.h*>) before this function completes, this function will automatically call **clearImages()** before throwing *ccTimeout::Expired*.

numImages `c_Int32 numImages() const;`

Returns the number of images added in the current stitching.

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

Notes

This is equal to the number of times **addImage()** has been called since the last call to **clearImages()**.

stitchedImage `const ccPelBuffer_const<c_UInt8>& stitchedImage() const;`

Returns the stitched result image.

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

Notes

The stitched image and mask image are both bound with the same image window and client transform.

See **init()** for information on how the window and client transform are determined.

stitchedMaskImage

```
const ccPelBuffer_const<c_UInt8>& stitchedMaskImage()
                                                    const;
```

Returns the stitched result image mask with pixel values either 0 or 255. A 0 specifies masked-out pixels (the corresponding image result pixels should be ignored). A 255 identifies unmasked (effective) result pixels.

Throws

ccImageStitchDefs::NotInitialized
if **isInitialized()** returns *false*

Notes

The stitched image and mask image are both bound with the same image window and client transform.

See **init()** for information on how the window and client transform are determined.

ccImageStitchDefs

```
#include <stitch.h>

class ccImageStitchDefs;
```

A name space that holds enumerations and constants used with the Image Stitching tool. See **ccImageStitch.**

Enumerations

OutputMode

```
enum OutputMode
```

This enumeration defines constants that specify the mode for computing the pixel value in a stitched image result. The tool supports two modes: *Blend* mode and *Overwrite* mode.

Value	Meaning
eBlend = 0x1	Specifies *Blend* mode
eOverwrite = 0x2	Specifies *Overwrite* mode

ccImagingDevice

```
#include <ch_cvl/vpimgdev.h>

class ccImagingDevice : public ccUnknownFG,
   public ccUnknownFeatureAccess ;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes an Imaging Device (a software module provided by a third-party manufacturer that provides access to an imaging device such as a camera or frame grabber from that manufacturer). There is one instance of this class for each enabled Imaging Device. You use the static **get()** function to obtain a specific instance.

```
ccImagingDevice& myDevice = ccImagingDevice::get(0);
```

Calling the **name()** function (a member of **ccBoard**, from which this class is derived) returns the name of the Imaging Device,

Constructors/Destructors

A single instance of this class is created automatically for each Imaging Device connected to your system.

Public Member Functions

count

```
static c_Int32 count();
```

Returns the number of Imaging Device present in the system.

get

```
static ccImagingDevice& get(c_Int32 i = 0);
```

Returns a **ccImagingDevice** object that represents the specified device. If multiple Imaging Devices are enabled on your system, the index order of the devices is undefined. You can use the **name()** function to identify the device returned by this function.

Parameters

i The index of the device to get.

Throws

ccBoard::BadParams

i is less than zero or greater than or equal to **count()**.

Notes

For any given index value, the object returned by this function and the object returned by calling **ccBoard::get()** with the same index may not refer to the same device, since **ccBoard::get()** returns references to *all* devices, such as Cognex frame grabbers, not just Imaging Devices.

serialNumber

```
ccCvlString serialNumber() const;
```

Returns the serial number of the Imaging Device. The format of the returned string is specific to the Imaging Device type.

executeCommand

```
void executeCommand(ccCvlString featureName);
```

Executes the command as defined by the imaging device. Refer to the documentation supplied with your Imaging Device for information about supported commands.

Parameters

featureName The command to issue.

Throws

ccImagingDevice::FeatureError

nodeName was not a valid node name.

readValue

```
ccCvlString readValue(ccCvlString featureName);
```

Returns the value associated with the specified feature for this Imaging Device. For information about the supported features and values, refer to the documentation supplied with your imaging device.

Parameters

featureName The name of the feature.

Notes

Use this function to access features that are not implemented directly in CVL. Use the standard CVL acquisition fifo interface to view exposure, contrast, brightness, region of interest, and trigger mode..

Throws

ccImagingDevice::FeatureError

featureName was not a valid feature name.

writeValue

```
void writeValue (ccCvlString featureName,
     ccCvlString value);
```

Sets the value associated with the specified feature for this Imaging Device. For information about the supported features and values, refer to the documentation supplied with your imaging device.

Parameters

featureName The name of the feature to set.

value The value fo set for this feature.

Throws

ccImagingDevice::FeatureError
featureName was not a valid feature name.

Notes

Use this function to access features that are not implemented directly in CVL. Use the standard CVL acquisition fifo interface to set exposure, contrast, brightness, region of interest, and trigger mode..

ccImagingDevice

ccImageWarp

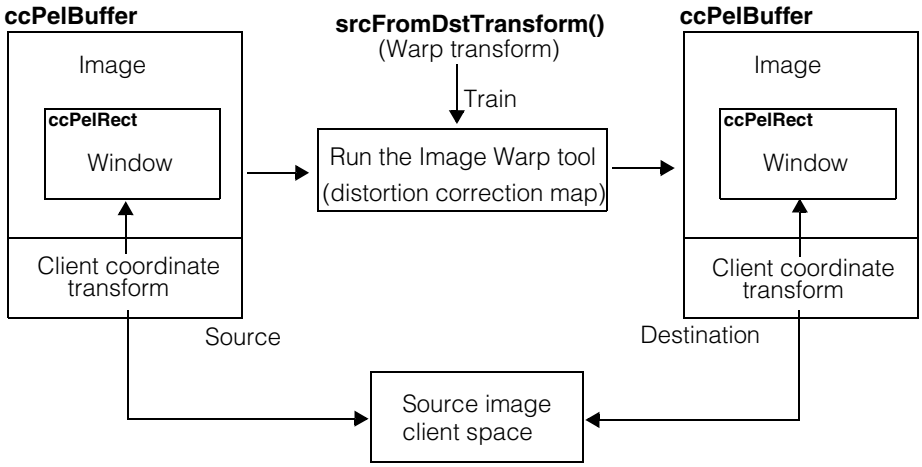
```
#include <ch_cvl/imgwarp.h>

class ccImageWarp;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class encapsulates a tool for warping images through 2-D transforms.



The tool can be used in one of following two modes:

General:

In this mode, you may specify any transform as the warp transform. This mode is selected when you call the function **srcFromDstTransform()** with a **cc2XformBase** object as its only argument.

Using calibration:

In this mode, a warp transform is set to a transform computed using the provided calibration transform plus additional information. This mode is selected when you call the function **srcFromDstTransform()** with a **cc2XformCalib2** object as one of its arguments. This mode is used to remove perspective and/or lens distortion from images.

These two modes differ only in following ways:

- How the warp transform is set.
- How the client transform of the warped image is set.

Constructors/Destructors

ccImageWarp

```
ccImageWarp();
ccImageWarp(const ccImageWarp &rhs);
~ccImageWarp();
```

- `ccImageWarp();`

  Default constructor.

- `ccImageWarp(const ccImageWarp &rhs);`

  Copy constructor.

- `~ccImageWarp();`

  Destructor.

Enumerations

Mode `enum Mode;`

Value	Meaning
eGeneral = *0x1*	You specify the warp transform.
eUsingCalibration = *0x2*	The warp transform is computed from a **cc2XformCalib2** transform and other parameters you provide.

UndistortMode `enum UndistortMode;`

Value	Meaning
eUndistortPerspective = 0x1	Correct perspective distortion. (Distortion that depends on the position and orientation of the object plane relative to the camera).
eUndistortRadial = 0x2	Correct radial distortion. (Barrel distortion or pincushion distortion).
eUndistortXYRotation = 0x4	Correct rotational distortion of the x-y axes in the image plane.
eUndistortDefault = *eUndistortPerspective* | *eUndistortRadial* | *eUndistortXYRotation*	

Operators

operator== `bool operator==(const ccImageWarp& other) const;`

Return true if this object is equal to *other*; return false otherwise.

Parameters

other Another **ccImageWarp** object to compare with this one.

operator= `ccImageWarp &operator=(const ccImageWarp &rhs);`

Assignment operator. Make this object a copy of *rhs*.

Parameters

rhs The object to copy.

Public Member Functions

srcRect

```
void srcRect(const ccPelRect& rect);

const ccPelRect& srcRect() const;
```

Defines a rectangular area (window) within the source image. See the **ccPelRect** reference page.

- `void srcRect(const ccPelRect& rect);`

  Sets the source image window. The object is set to the untrained state.

  Parameters

  rect The new source image window.

  Throws

  ccImageWarpDefs::BadWindow
  If *rect* is a null rectangle.

- `const ccPelRect& srcRect() const;`

  Returns the source image window.

srcFromDstTransform

```
cc2XformBasePtrh_const srcFromDstTransform() const;

void srcFromDstTransform(const cc2XformBase& xform);

void srcFromDstTransform (
  const cc2XformCalib2& fromPhysicalCalibrationXform,
  const cc2Vect& pt,
  c_UInt32 undistortMode = ccImageWarp::eUndistortDefault);
```

- `cc2XformBasePtrh_const srcFromDstTransform() const;`

  Returns the current warp transform. If the following setter was last used, the transform that was set is returned. If the second setter was last used, the warp transform computed from *fromPhysicalCalibrationXform*, *pt*, and *undistortMode* is returned.

- `void srcFromDstTransform(const cc2XformBase& xform);`

  Sets a new warp transform that maps destination image coordinates to source image coordinates. The object is set to untrained. Mode is set to *eGeneral*.

  This setter is provided as a way to use a **cc2XformLinear** or **cc2XformPoly** as the Warp transform.

  Parameters

  xform The new warp transform.

- ```
  void srcFromDstTransform (
    const cc2XformCalib2& fromPhysicalCalibrationXform,
    const cc2Vect& pt,
    c_UInt32 undistortMode);
  ```

  Compute and set the warp transform using the arguments provided. The object is set to untrained. Mode is set to *eUsingCalibration*.

  The warp transform that is computed can be retrieved using the **srcFromDstTransform()** getter above.

  Parameters

  fromPhysicalCalibrationXform
  A nonlinear calibration transform from physical space to image space. For information about this transform, see the **cc2XformCalib2** reference page.

  pt A 2-D point within the source window. The scale at this point in the source image is used as the scale for the entire destination image. This is done so that features in the undistorted image are approximately the same size as in the distorted image.

  undistortMode Specifies which distortions (perspective, lens, and so on) should be removed. *undistortMode* can be any combination of the modes defined in the **ccImageWarp::UndistortMode** enum.

  Note: If the calibration was done using a single image, only *eUndistortDefault* is allowed here.

  Throws

  cc2XformCalib2Defs::BadParams
  If *fromPhysicalCalibrationXform* has been initialized with a single correspondence and *undistortMode* != *eUndistortDefault*.
  ```

calibrationTransform

```
cc2XformCalib2Ptrh_const calibrationTransform() const;
```

In *eUsingCalibration* mode this function returns the **cc2XformCalib2** calibration transform. (See the *fromPhysicalCalibrationXform* parameter in **srcFromDstTransform()** above). In *eGeneral* mode it returns an identity transform.

pt

```
cc2Vect pt() const;
```

In *eUsingCalibration* mode this function returns the point used to create the warp transform. (See the *pt* parameter in **srcFromDstTransform()** above). In *eGeneral* mode it returns (0, 0).

undistortMode

```
c_UInt32 undistortMode() const;
```

In *eUsingCalibration* mode this function returns the undistort mode. (See the *undistortMode* parameter in **srcFromDstTransform()** above). In *eGeneral* mode it returns 0.

mode

```
ccImageWarp::Mode mode() const;
```

Returns the trained mode; *eUsingCalibration* or *eGeneral*.

accuracy

```
void accuracy(double threshold);

double accuracy() const;
```

Accuracy is the maximum allowed error, in source pixels, between a point's true position in the source image and its warped position in the destination image. Smaller accuracy values will typically make the transform more accurate, but will increase the time and memory space required for execution.

The training algorithm maps points from the source image to the destination image and builds a mapping table it uses when the tool is run. More granularity in this table creates better accuracy and uses more memory. To check accuracy during training, pixels are transposed to the destination and then back to the source. If pixels transposed back to the source have a location error greater than *accuracy* pixels, the table granularity is increased until the accuracy limit is reached.

The default is 0.001 pels.

- `void accuracy(double threshold);`

  Sets a new accuracy value and sets the object to the untrained state.

Parameters

threshold The new accuracy.

Throws

ccImageWarpDefs::BadParams
If *threshold* is negative.

- `double accuracy() const;`

  Returns the current accuracy value. The default is 0.001 pixels.

setDstClientFromImageXform

```
void setDstClientFromImageXform(
   bool setDstClientFromImageXform);

bool setDstClientFromImageXform() const;
```

- `void setDstClientFromImageXform(`
  `bool setDstClientFromImageXform);`

  Sets whether or not to compute and set the destination image's **clientFromImageXform()** as described in **warp** on page 1750.

  If your application acquires multiple images of the same scene using the same physical and optical configuration, you can optimize its speed by computing the **clientFromImageXform()** transform once, then applying that value to each warped image. Setting this member to *false* prevents the tool from recomputing the transform for each image, enabling the optimization.

  This member is set to *true* in a default-constructed **ccImageWarp** object.

  Parameters

  setDstClientFromImageXform
  If true, the transform will be computed.

- `bool setDstClientFromImageXform() const;`

  Returns whether or not the tool is computing the transform.

dstRect

```
void dstRect(const ccPelRect& rect);
const ccPelRect& dstRect() const;
```

- `void dstRect(const ccPelRect& rect);`

  Sets the destination window and sets the object to the untrained state.

  Parameters

  rect The new destination window.

  Throws

  ccImageWarpDefs::BadWindow
  If *rect* is a NULL rectangle.

- `const ccPelRect& dstRect() const;`

  Returns the current destination window.

trainedSrcRect

```
const ccPelRect& trainedSrcRect() const;
```

Returns the trained source window. If the object is not trained a null rectangle is returned. The size of the trained source window is the size of the overlap between the source window (**srcRect()**) and the destination window (**dstRect()**).

trainedDstRect

```
const ccPelRect& trainedDstRect() const;
```

Returns the trained destination window. If the object is not trained a null rectangle is returned. The size of the trained destination window is the size of the overlap between the source window (**srcRect()**) and the destination window (**dstRect()**).

dstMask

```
const ccPelBuffer<c_UInt8>& dstMask() const;
```

Returns the destination mask.

When you train the Image Warp tool it may not be possible for the tool to map every pixel from the source to the destination. To indicate which pixels are mapped and which are not, the tool creates a *destination mask*. Mask pixels set to (255) indicate pixels that are mapped. Mask pixels set to (0) indicate pixels that are not mapped.

The destination mask window is the same size and location as the trained destination window (**trainedDstRect()**).

train `void train();`

Trains this Image Warp tool. If the destination window is a null rectangle the entire source window is warped.

If source window is a null rectangle the source window is computed by mapping the destination window into source image.

If neither the source nor destination windows are null rectangles, source and destination windows are computed as the intersection of the two windows.

If the object is trained, it will be untrained first and then retrained.

Throws

ccImageWarpDefs::BadWindow

If the source and destination windows are null rectangles,
or if the destination window mapped to the source image coordinates does not overlap the source window,
or if the source window mapped to the destination image coordinates does not overlap the destination window,
or if any of the (x,y) coordinates in the computed source or the destination is greater than or equal to 2^{31} or less than -2^{31}

Notes

The trained source and destination windows can be accessed by calling **trainedSrcRect()** and **trainedDstRect()**. Trained source and destination windows are computed using all boundary points of mapped source and destination.

untrain `void untrain();`

Untrains this object and releases memory.

isTrained `bool isTrained() const;`

Returns true if this object is currently trained, and returns false otherwise.

warp

```
void warp(
  const ccPelBuffer_const<c_UInt8>& src,
  ccPelBuffer<c_UInt8>& dst);
```

Calling this function runs the tool which transforms the source into the destination using the warp transform. The destination client from image transform maps destination pels to the source client space.

This object must be trained before calling **warp()**.

The window of the supplied destination image must contain the trained destination window. If the destination is unbound it will be created using the trained destination window.

Parameters

src The source image.

dst The destination image.

Throws

ccImageWarp::BadImage
If the source image is not bound.

ccImageWarp::NotTrained
If this object is not trained.

ccImageWarp::BadWindow
If the window of the supplied source image does not contain the trained source window,

or if the window of the supplied destination image is not contained in the trained destination window.

Notes

If **setDstClientFromImageXform()** is *true*, **clientFromImageXform()** of the destination pel buffer is set according to following rules:

eGeneral mode;

destination **clientFromImageXform()** =
(source **clientFromImageXform()**) * (**srcFromDstTransform()** provided).

eUsingCalibration mode;

(A) destination **clientFromImageXform()** =
(B) inverse of **fromPhysicalCalibrationXform()** provided
(C) * (**clientFromImageXform()** of source image)
(D) * (**srcFromDstXform()** computed internally)

For example, using the names A, B, C, D as shown above, for any point p in the destination, the following holds true:

A*p = B*C*D*p

ccImageWarp1D

```
#include <ch_cvl/imgwrp1d.h>

class ccImageWarp1D;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class encapsulates a specialized image warper intended for rectifying images acquired from a line scan camera. The functionality of this warper is equivalent to the functionality of the 2D image warper (encapsulated in the class **ccImageWarp** on page 1741) were that warper to be trained using a transformation that was linear in the y-direction.

The 1D warper can warp images of any height and any y-offset, as long as the width is the same as that used for training.

For additional information on this class, thee the file *ch_cvl/imgwrp1d.h*.

Constructors/Destructors

ccImageWarp1D

```
ccImageWarp1D();

ccImageWarp1D(const ccImageWarp1D &rhs);
```

- `ccImageWarp1D();`

  Default-constructs a **ccImageWarp1D** with the following values:

  - **srcFromDstTransform**: default-constructed **cc2XformLinear**.
  - **srcRect**: default-constructed **ccPelRect**.
  - **dstSpan**: default-constructed **ccPelSpan**.

- `ccImageWarp1D(const ccImageWarp1D &rhs);`

  Copy constructor.

- `~ccImageWarp1D();`

  Destructor.

Public Member Functions

srcRect

```
void srcRect(const ccPelRect& rect);
const ccPelRect &srcRect() const;
```

Defines a rectangular area (window) within the source image.

- `void srcRect(const ccPelRect& rect);`

  Sets the source image window. The object is set to the untrained state.

  Parameters

  rect The new source image window.

  Throws

  ccImageWarpDefs::BadWindow
  If *rect* is a null rectangle.

- `const ccPelRect& srcRect() const;`

  Returns the source image window.

dstSpan

```
void dstSpan(const ccPelSpan& span);
const ccPelSpan &dstSpan() const;
```

- `void dstSpan(const ccPelSpan& span);`

  Sets the destination span for the unwarp. Calling this overload untrains the warper.

  Since the 1D warper may be used to unwarp images of any height, the output region is specified as a span rather than a rectangle.

  Parameters

  span The span.

  Throws

  ccImageWarp1DDefs::BadWindow
  span is a null span.

- `const ccPelSpan &dstSpan() const;`

  Returns the current destination span for the unwarp.

srcFromDstTransform

```
void srcFromDstTransform(const cc2XformBase& xform);

cc2XformBasePtrh_const srcFromDstTransform() const;
```

- `void srcFromDstTransform(const cc2XformBase& xform);`

  Sets the transformation used for the warping. In most cases, this is the transform produced by calling **cfCalibrateLineScanCamera()**.

  Calling this overload untrains the warper.

  Parameters

  xform The transform.

- `cc2XformBasePtrh_const srcFromDstTransform() const;`

  Returns the current warping transform.

setDstClientFromImageXform

```
void setDstClientFromImageXform(
   bool setDstClientFromImageXform);

bool setDstClientFromImageXform() const;
```

- ```
  void setDstClientFromImageXform(
     bool setDstClientFromImageXform);
  ```

  Sets whether or not to compute and set the destination image's **clientFromImageXform()**.

  If your application acquires multiple images of the same scene using the same physical and optical configuration, you can optimize its speed by computing the **clientFromImageXform()** transform once, then applying that value to each warped image. Setting this member to *false* prevents the tool from recomputing the transform for each image, enabling the optimization.

  Parameters

  setDstClientFromImageXform

  If true, the transform will be computed.
  ```

- `bool setDstClientFromImageXform() const;`

  Returns whether or not the tool is computing the transform.

accuracy

```
void accuracy(double threshold);
double accuracy() const;
```

- `void accuracy(double threshold);`

  Accuracy is the maximum allowed error, in source pixels, between a point's true position in the source image and its warped position in the destination image. Smaller accuracy values will typically make the transform more accurate, but will increase the time and memory space required for execution.

  The training algorithm maps points from the source image to the destination image and builds a mapping table it uses when the tool is run. More granularity in this table creates better accuracy and uses more memory. To check accuracy during training, pixels are transposed to the destination and then back to the source. If pixels transposed back to the source have a location error greater than *accuracy* pixels, the table granularity is increased until the accuracy limit is reached.

  The default is 0.001 pels.

  Parameters

  threshold The threshold in pixels.

  Throws

  ccImageWarp1DDefs::BadParams
  threshold is negative.

- `double accuracy() const;`

  Returns the current accuracy threshold.

trainedSrcRect

`const ccPelRect &trainedSrcRect() const;`

Returns the trained source window. If the object is not trained a null rectangle is returned.

trainedDstSpan

`const ccPelSpan &trainedDstSpan() const;`

Returns the trained destination window. If the object is not trained a null rectangle is returned.

Notes

train `void train();`

Train the warper. If **dstSpan** is a null span, the whole source window is warped. If **srcRect** is a null rectangle, the source rect is computed by mapping **dstSpan** into source image coordinates. If both **srcRect** and **dstSpan** are not null, then **trainedSrcRect** and **trainedDstSpan** are computed as an intersection of **srcRect** and **dstSpan** (the source rect is computed in source image coordinates and the destination span is computed in destination image coordinates).

Notes

If the object is already trained, then it is first untrained and then retrained

The trained windows for src and dst can be accessed using **trainedSrcRect** and **trainedDstSpan**. **trainedSrcRect** and **trainedDstSpan** are computed using all boundary points of mapped source and destination.

Throws

ccImageWarp1DDefs::BadWindow

The source window and destination span are null, or the destination span when mapped into source image coordinates does not overlap the source window, or any of the (x,y) coordinates in the computed source or the destination is greater than or equal to 2^{31} or less than -2^{31}

untrain `void untrain();`

This function untrains the warper and releases memory.

isTrained `bool isTrained() const;`

Returns true if this is currently trained, false otherwise.

warp

```
void warp(const ccPelBuffer_const<c_UInt8>& src,
   ccPelBuffer<c_UInt8>& dst) const;

void warp(const ccPelBuffer_const<c_UInt8>& src,
   ccPelBuffer<c_UInt8>& dst,
   ccPelBuffer<c_UInt8>& dstMask) const;
```

- ```
  void warp(const ccPelBuffer_const<c_UInt8>& src,
     ccPelBuffer<c_UInt8>& dst) const;
  ```

  Calling this function runs the tool which transforms the source into the destination using the warp transform. The destination client from image transform maps destination pels to the source client space.

  This object must be trained before calling **warp()**.

  The window of the supplied destination image must contain the trained destination span. If the destination is unbound it will be created using the trained destination window.

  Parameters

  src The source image. *src* may be of an arbitrary height and y-offset.

  dst The destination image.

  Notes

  The x extents of the supplied dst image must contain the trained dst span. If dst is unbound it will be created using trained dst span.

  If setDstClientFromImageXform() is *true*, clientFromImageXform() of the destination pel buffer is set according to following rule:

  destination **clientFromImageXform()** =
  (source **clientFromImageXform()**) * (**srcFromDstTransform()** provided).

  Throws

  ccImageWarp1DDefs::BadImage
  If the source image is not bound.

  ccImageWarp1DDefs::NotTrained
  If this object is not trained.

  ccImageWarp1DDefs::BadWindow
  If the window of the supplied source image does not contain the trained source window,

  or if the x-span of the supplied destination span is not contained in the trained destination span.
  ```

- ```
  void warp(const ccPelBuffer_const<c_UInt8>& src,
     ccPelBuffer<c_UInt8>& dst,
     ccPelBuffer<c_UInt8>& dstMask) const;
  ```

  Calling this function runs the tool which transforms the source into the destination using the warp transform. The destination client from image transform maps destination pels to the source client space.

  Compute and store the destination mask where care pixels (255) in *dstMask* correspond to the dst coordinates that are mapped to the valid src pixels and don't care pixels (0) correspond to the coordinates in the dst image that do not have valid mapping to the src image.

  This object must be trained before calling **warp()**.

  The window of the supplied destination image must contain the trained destination span. If the destination is unbound it will be created using the trained destination window.

  Parameters

  src The source image. *src* may be of an arbitrary height and y-offset.

  dst The destination image.

  dstMask The mask image.

  Notes

  The x extents of the supplied dst image must contain the trained dst span. If dst is unbound it will be created using trained dst span.

  If setDstClientFromImageXform() is *true*, clientFromImageXform() of the destination pel buffer is set according to following rule:

  destination **clientFromImageXform()** =
  (source **clientFromImageXform()**) * (**srcFromDstTransform()** provided).

  Throws

  ccImageWarp1DDefs::BadImage
  If the source image is not bound.

  ccImageWarp1DDefs::NotTrained
  If this object is not trained.

  ccImageWarp1DDefs::BadWindow
  If the window of the supplied source image does not contain the trained source window, if the x-span of the supplied destination span is not contained in the trained destination span, or if the x-span of the supplied destination mask image is not contained in the trained destination span.
  ```

Static Functions

computeSrcFromDstTransform

```
static cc2XformBasePtrh_const computeSrcFromDstTransform (
   const cc2XformBase& fromPhysicalCalibrationXform,
   const cc2Vect& pt);
```

Compute warping transform using the arguments provided: that

- *fromPhysicalCalibrationXform* is a calibration transform from physical space.
- *pt* is a 2d point. It is used to set the scale so that in the features in undistorted image are approximately of same size as in the distorted image.

Parameters

fromPhysicalCalibrationXform
The calibration transform.

pt A point that defines the x- and y-scale.

Operators

operator=

```
ccImageWarp1D &operator=(const ccImageWarp1D &rhs);
```

Assignment operator.

operator==

```
bool operator==(const ccImageWarp1D& other) const;
```

Return true if this object is equal to the supplied object, false otherwise.

Parameters

other The object to compare to this one.

ccIndexChain

```
#include <ch_cvl/chain.h>

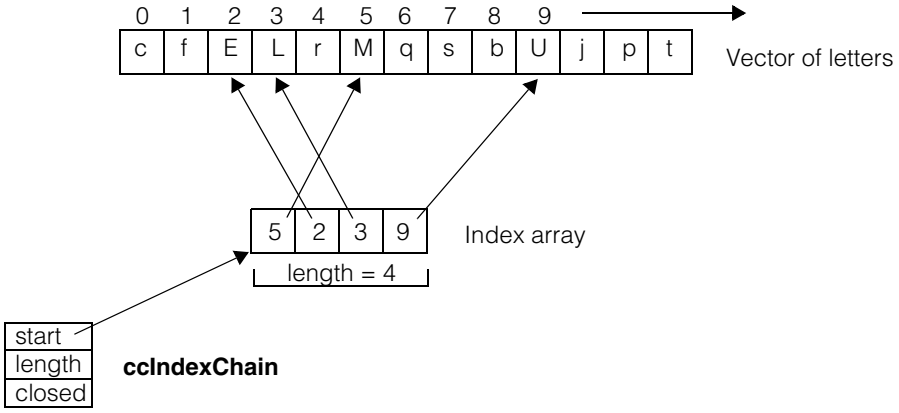
class ccIndexChain;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class encapsulates the parameters necessary to identify a chain of related items in an array, without altering the array. It includes a pointer to an index array that you create independently, the length of the index array, and a boolean that indicates if the chain is closed. In a closed chain, the first and last items are adjacent.

The following is an example of using a **ccIndexChain** to identify the uppercase characters in an array of characters. The uppercase characters form a chain. *start* points to an array that holds indices into the character array above where the uppercase letters are stored. *length* specifies that there are four characters in the chain, and *closed* will be false in this example.



ccIndexChain objects are used by the Edge tool to identify found edgelets that are related. See the **cfEdgeDetect()** reference page and also the **cfFilterEdgeletChains()** reference page other examples.

Constructors/Destructors

ccIndexChain

```
ccIndexChain();

ccIndexChain(
  c_Int32 *start,
  c_Int32 len,
  bool closed);
```

- `ccIndexChain();`

  Default constructor. Sets the following default values:

Parameter	Default Value
start	0
len	0
closed	false

- ```
  ccIndexChain(
    c_Int32 *start,
    c_Int32 len,
    bool closed);
  ```

  Constructor.

  Parameters

  start — A pointer to the beginning of an index array that holds indices into another array. The index array specifies the chain.

  len — The chain length.

  closed — True is the chain is closed, false otherwise.

Public Member Functions

start

```
const c_Int32 *start () const;
```

Returns a pointer to the start of an index array.

length

```
c_Int32 length () const;
```

Returns the number of indices in the chain. This is also the size of the index array.
```

closed

```
bool closed () const;
```

Returns true if a closed chain is specified, and false otherwise.

ccIndexChain

ccIndexChainList

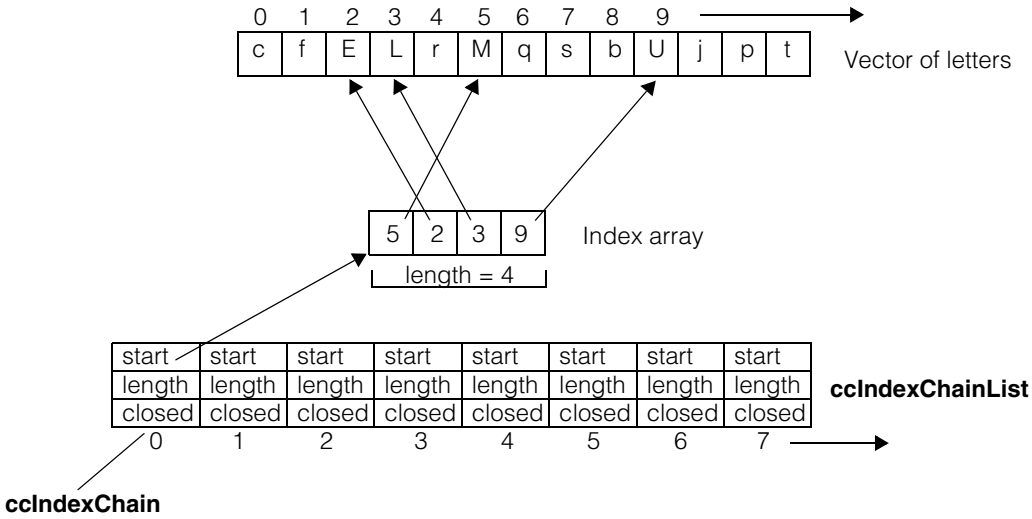
```
#include <ch_cvl/chain.h>

class ccIndexChainList
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class holds an array of **ccIndexChain** objects, a chain list. See the following diagram.



See the **ccIndexChain** reference page for information about this class.

Constructors/Destructors

ccIndexChainList

```
ccIndexChainList ();
~ccIndexChainList ();
ccIndexChainList (const ccIndexChainList&);
```

- `ccIndexChainList ();`

  Default constructor.

- `~ccIndexChainList ();`

  Destructor.

- `ccIndexChainList (const ccIndexChainList&);`

  Copy constructor.

Operators

operator= `ccIndexChainList& operator= (const ccIndexChainList&);`

Efficient deep copy.

operator[] `const ccIndexChain& operator[] (int i) const;`

Returns chain *i*.

Parameters

i The chain index.

Throws

ccIndexChainList::OutOfBounds
If *i* is not a valid chain index.

Public Member Functions

size `int size () const;`

Returns the number of chains on the list.

ccInputLine

```
#include <ch_cvl/pio.h>

class ccInputLine;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class describes a single input line. You obtain a **ccInputLine** by calling the **inputLine()** member of the class that describes your hardware platform. A value of true for an input line generally designates a high state (+5V); a value of false generally designates a low state (0V).

Notes

Do not attempt to construct a **ccInputLine** directly.

Enumerations

ceTrigCondition This enumeration specifies the transitions of an input line that cause a trigger.

Value	Meaning
ceHighToLow	Transition from 1 to 0 causes a trigger.
ceLowToHigh	Transition from 0 to 1causes a trigger.
ceBoth	Any transition causes a trigger.
ceNone	No transition causes a trigger.

Public Member Functions

get

```
bool get() const;
```

Returns the current state of the line represented by this **ccInputLine**. True designates a high state. False designates a low state.

Throws

ccParallelIO::NotEnabled

The input line object was not constructed correctly.

lineNumber `c_Int32 lineNumber() const;`

Returns the logical line number associated with this **ccInputLine**. The value returned is determined by the configuration set by **ccParallelIO::setIOConfig()**.

Throws

ccParallelIO::NotEnabled
The input line object was not constructed correctly.

Notes

See the documentation for the class that describes your specific hardware platform for the equivalence between logical line numbers and physical lines.

enable `void enable(bool value);`

Enables the input line associated with this **ccInputLine**.

Parameters

value True enables the line. False disables the line.

Throws

ccParallelIO::CannotEnable
This **ccInputLine** cannot be enabled. If you are using a hardware platform that supports bidirectional lines, then it may not be possible to enable an input line if other lines in the same group have been configured as output lines.

Notes

If this input line was set up as a trigger, and *value* is false, any callback function associated with this line is uninstalled.

enabled `bool enabled() const;`

Returns true if this line is enabled. Returns false if this line not enabled.

canEnable `bool canEnable() const;`

Returns true if the input line associated with this **ccInputLine** can be enabled. Returns false if **enable(true)** would throw an error.

Throws

ccParallelIO::NotEnabled
The input line object was not constructed correctly.

enableCallback

```
ccInputLine::Token enableCallback(
  const ccCallback1Ptrh& action,
  void* arg,
  cePriority priority,
  ceTrigCondition polarity);
```

Returns a **ccInputLine::Token** object that represents a callback function. When the token is destroyed, either by deletion, by going out of scope, or by assignment to another token, the callback is disabled. The callback remains enabled for the lifetime of the token.

To verify that **enableCallback()** completed successfully, cast the token to a boolean. True indicates successful completion. False indicates a failure. Multiple callbacks on the same physical input line may have requirements such as using a common polarity.

Parameters

action — A pointer to the action routine to be executed when a transition is detected on the line.

If you want to remove a callback function but leave the input line enabled, assign the callback token to a default-constructed token, for example using:

```
myToken = ccInputLine::Token();
```

This call will unregister the callback associated with *myToken*.

Assigning NULL to this parameter does not disable the callback.

Disabling the input line does not unregister callbacks, it will only temporarily deactivate them. The callbacks will become active again when the line is enabled. Callbacks remain registered throughout the life of their associated token. To unregister a callback, use the above method.

See **ccCallback** and **ccCallback1** for further information on the use of callbacks.

arg — This argument is passed to the callback function when it is invoked.

priority — The thread priority of the callback function. See **cfCreateThread()**

polarity — The condition on which this input line should trigger. Must be one of the following:

ceInputLine::ceHighToLow
ceInputLine::ceLowToHigh
ceInputLine::ceBoth

Throws

ccParallelIO::BadParams
One of the parameters was not valid.

ccParallelIO::CannotEnable
This **ccInputLine** cannot be enabled or may be in use.

Notes

Callbacks are supported on Cognex frame grabbers as follows:

- On the MVS-8500 series, callbacks are supported on any bidirectional line that has been configured as an input line.

See the sample *pio.cpp*, provided in the *%VISION_ROOT%\sample\cvl* directory, for code demonstrating how to create and use input callbacks.

ccInterpSpline

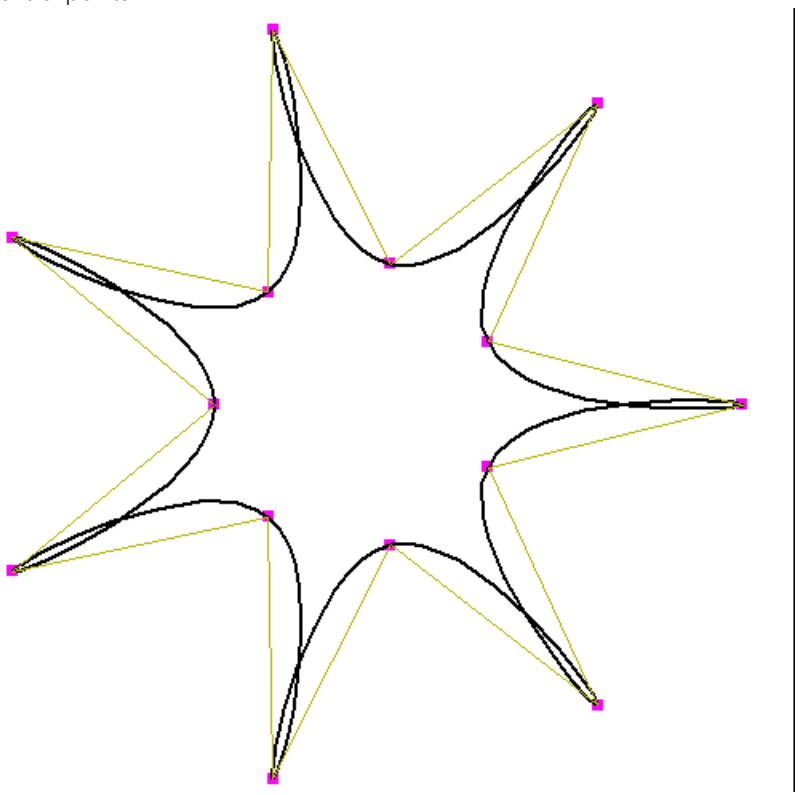
```
#include <ch_cvl/spline.h>

class ccInterpSpline : public ccCubicSpline;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccInterpSpline** class is an implementation of a 2D cubic C^2 interpolation spline, a type of cubic B-Spline in which the spline curve interpolates (passes through) all of the spline control points.



The above figure shows an example of a cubic C^2 interpolation spline.

Properties of Interpolation Splines

Interpolation of all of the control points is often a desirable property when using splines for modeling or design. Each component Bezier curve lies between two successive control points, and the spline is twice differentiable at the junction points (unless there are duplicated knots), and infinitely differentiable everywhere else.

The minimum number of spline control points needed to specify a non-empty **ccInterpSpline** is two for an open spline, and three for a closed spline. In other words, a **ccInterpSpline** is empty if it has fewer than two control points; a closed **ccInterpSpline** is empty if is has fewer than three control points.

End Conditions

Tangent vectors at the junction points of a cubic C^2 interpolation spline are determined by the B-spline differentiability constraints. Open cubic C^2 interpolation splines require the specification of end conditions, which determine the tangent vectors at the endpoints of the spline. There are several options:

Condition	Endpoint Tangents
Bessel	Chosen as tangent vectors to the interpolating parabolas through the first three and last three control points.
Quadratic	Chosen so the first and last cubic Bezier curves are quadratic (cubic coefficients are zero).
Natural	Chosen so second derivatives of x(*t*) and y(*t*) vanish at the endpoints; the spline exhibits low curvature near the ends.
Clamped	Explicitly specified.

Bessel end conditions are a good default choice.

Comparison of de Boor Splines and C^2 Interpolation Splines

The three properties described for cubic de Boor splines do not hold for cubic C^2 interpolation splines.

Cubic Spline Property	de Boor Splines	C^2 Interpolation Splines
Curve must lie within the convex hull of the spline control points	Yes	No
Number of component Bezier curves affected by moving a spline control point	Limited number	All are affected, however those farther away from the moved control point are affected less.
Assigning any non-negative weight to control points produces a valid spline	Yes	N/A

Interpolating splines with one or more zero intervals are not well defined. As a fallback, the cubic Bezier curve representation computed for such an interpolating spline is the control polygon itself, whether it be open or closed. It is generally advisable to avoid zero intervals with interpolating splines.

Note Avoiding zero-length intervals also means that successive interpolation points should not be coincident when parameterizations based on distance between interpolation points are in effect.

Computing Bezier control points from the spline control points is a more expensive operation for cubic C^2 interpolation splines than for cubic de Boor splines. However, once the Bezier curves are computed (and cached), you can query or render a cubic C^2 interpolation spline as quickly as a cubic de Boor spline of similar size.

Note For in-depth information on the theory and use of 2D cubic Bezier curves and splines, see any textbook on the subject, such as *Curves and Surfaces for Computer Aided Geometric Design* by Gerald Farin, Second Edition, Academic Press, 1990, ISBN 0-12-249051-7.

Note All methods defined for this class leave the **ccInterpSpline** object unchanged when they throw an error.

Enumerations

EndConditions `enum EndConditions;`

See *End Conditions* on page 1772 for details.

Value	Meaning
eBessel	Endpoint tangents are chosen as tangent vectors to the interpolating parabolas through the first three and last three control points.
eQuadratic	Endpoint tangents are chosen so the first and last cubic Bezier curves are quadratic (cubic coefficients are zero).
eNatural	Endpoint tangents are chosen so the second derivatives of x(*t*) and y(*t*) vanish at the endpoints; the spline exhibits low curvature near the ends.
eClamped	Endpoint tangents are explicitly specified.

Operators

operator== `bool operator==(const ccInterpSpline &rhs) const;`

Returns true if and only if this **ccInterpSpline** is equal to *rhs*. Two splines are equal if all of their defining parameters are identical (no tolerance is used).

Parameters

rhs — The other **ccInterpspline**.

Notes

Splines that describe identical curves are not necessarily equal.

operator!= `bool operator!=(const ccInterpSpline &rhs) const;`

Returns the opposite truth value to **operator==()**.

Parameters

rhs — The other **ccInterpspline**.

Constructors/Destructors

```
explicit
ccInterpSpline(bool isClosed = false,
  IntervalMode intervalMode = eUniform,
  EndConditions endConditions = eBessel);

explicit ccInterpSpline(
  const cmStd vector<cc2Vect> &ctrlPts,
  bool isClosed = false,
  IntervalMode intervalMode = eUniform,
  EndConditions endConditions = eBessel);
```

- ```
  explicit
  ccInterpSpline(bool isClosed = false,
    IntervalMode intervalMode = eUniform,
    EndConditions endConditions = eBessel);
  ```

  Constructs an empty interpolating B-spline with no control points, with the given open/closed state, interval mode, and end conditions. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

isClosed	The open/closed state of this **ccInterpSpline** (default = open).
intervalMode	The interval mode. Must be one of the following: *ccCubicSpline::eUniform (default)* *ccCubicSpline:eChordLength* *ccCubicSpline::eCentripetal* *ccCubicSpline::eFixed*
endConditions	Conditions that determine the tangent vectors at the endpoints of the spline. Must be one of the following: *ccInterpSpline::eBessel* (default) *ccInterpSpline::eQuadratic* *ccInterpSpline::eNatural* *ccInterpSpline::eClamped*

- ```
  explicit ccInterpSpline(
    const cmStd vector<cc2Vect> &ctrlPts,
    bool isClosed = false,
    IntervalMode intervalMode = eUniform,
    EndConditions endConditions = eBessel);
  ```

  Constructs an interpolating B-spline with the given control points, open/closed state, interval mode, and end conditions. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

ctrlPts	The vector of control points.
isClosed	The open/closed state of this **ccInterpSpline** (default = open).
intervalMode	The interval mode. Must be one of the following: *ccCubicSpline::eUniform (default)* *ccCubicSpline:eChordLength* *ccCubicSpline::eCentripetal* *ccCubicSpline::eFixed*
endConditions	Conditions that determine the tangent vectors at the endpoints of the spline. Must be one of the following: *ccInterpSpline::eBessel* (default) *ccInterpSpline::eQuadratic* *ccInterpSpline::eNatural* *ccInterpSpline::eClamped*

endConditions

```
EndConditions endConditions() const;

void endConditions(EndConditions endConditions);
```

- `EndConditions endConditions() const;`

  Returns the end conditions of this spline, which determine the tangent vectors at the endpoints of the spline. See **EndConditions** on page 1774.

- `void endConditions(EndConditions endConditions);`

  Sets the end conditions of this spline.

  Parameters

endConditions	Conditions that determine the tangent vectors at the endpoints of the spline. Must be one of the following: *ccInterpSpline::eBessel* (default) *ccInterpSpline::eQuadratic* *ccInterpSpline::eNatural* *ccInterpSpline::eClamped*

  Notes

  This value is only significant if this spline is open and has clamped end conditions. Otherwise the value is ignored.

startDeriv

```
cc2Vect startDeriv() const;
void startDeriv(const cc2Vect &v);
```

- `cc2Vect startDeriv() const;`

  Gets the derivative vector at the start point of this spline.

- `void startDeriv(const cc2Vect &v);`

  Sets the derivative vector at the start point of this spline.

  Parameters

  v The derivative vector.

  Notes

  This value is significant only if this spline is open and has clamped end conditions, otherwise the value is ignored.

endDeriv

```
cc2Vect endDeriv() const;
void endDeriv(const cc2Vect &v);
```

- `cc2Vect endDeriv() const;`

  Gets the derivative vector at the end point of this spline.

- `void endDeriv(const cc2Vect &v);`

  Get/set the derivative vector at the end point of this spline.

  Parameters

  v The derivative vector.

  Notes

  This value is only significant if this spline is open and has clamped end conditions, otherwise the value is ignored.

map

```
ccInterpSpline map(const cc2Xform& X) const;
```

Returns this ccInterpSpline mapped by *X*.

Parameters

X The transformation object.

Notes

This function maps the control point positions by *X* and leaves the weights and intervals unchanged. It may be desirable to invoke reparameterize() on the returned spline. See *Mapping Splines* and *Reparameterizing Splines* on page 1112 for details.

Note Several of **ccCubicSpline**'s setter/getter pairs have a virtual setter and a non-virtual getter. The getters must be explicitly exposed in derived classes, or they will be hidden by the setter overrides.

---

isClosed

```
bool isClosed() const;

virtual void isClosed(bool);
```

- `bool isClosed() const;`

  Returns true if this **ccInterpSpline** is closed, false if it is open.

- `virtual void isClosed(bool);`

  Sets the open/closed state of this **ccInterpSpline**.

  Parameters

  bool True sets this spline to closed, false sets it to open.

  See **ccCubicSpline::isClosed()** for more information.

---

controlPoint

```
cc2Vect controlPoint(c_Int32 idx) const;

virtual void controlPoint(c_Int32 idx,
  const cc2Vect &ctrlPt);
```

- `cc2Vect controlPoint(c_Int32 idx) const;`

  Gets the control point with the given index.

  Parameters

  idx The index.

- ```
  virtual void controlPoint(c_Int32 idx,
    const cc2Vect &ctrlPt);
  ```

  Sets the control point with the given index.
  ```

Parameters

idx The index.

ctrlPt The control point.

See **ccCubicSpline::controlPoint()** for more information.

controlPoints

```
const cmStd vector<cc2Vect> &controlPoints() const;

virtual void controlPoints(
  const cmStd vector<cc2Vect> &ctrlPts);
```

- `const cmStd vector<cc2Vect> &controlPoints() const;`

  Gets the entire vector of control points.

- ```
  virtual void controlPoints(
    const cmStd vector<cc2Vect> &ctrlPts);
  ```

  Sets the entire vector of control points.

  Parameters

  ctrlPts The vector of control points.

  See **ccCubicSpline::controlPoints()** for more information.

insertControlPoint

```
virtual void insertControlPoint(c_Int32 idx,
  const cc2Vect &point);
```

Inserts a new control point before the point with the given index.

Parameters

idx The index.

point The control point.

Throws

ccShapesError::BadIndex

idx is less than 0, or greater than **numControlPoints()** before the insertion.

See **ccCubicSpline::insertControlPoint()** for more information.
```

removeControlPoint

```
virtual void removeControlPoint(c_Int32 idx);
```

Remvoes the control point with the given index.

Parameters

idx The index.

Throws

ccShapesError::BadIndex

idx is less than 0, or greater than **numControlPoints()** before the removal.

See **ccCubicSpline::removeControlPoint()** for more information.

reparameterize

```
virtual void reparameterize();
```

Recomputes all intervals based on the current control points and interval mode.

See **ccCubicSpline::reparameterize()** for more information.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this interpolation spline.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this **ccInterpSpline**. See **ccShape::reverse()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this **ccInterpSpline** mapped by *X*.

Parameters

X The transformation object.

See **ccShape::mapShape()** for more information.

ccIO8500l

```
#include <ch_cvl/pioconfg.h>

class ccIO8500l : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8500L frame grabber when used with cable 300-0390 and the pass-through TTL connection module, P/N 800-5818-1.

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams

The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines();
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines();
```

Returns the number of output lines for this hardware configuration.

ccIO8501

```
#include <ch_cvl/pioconfg.h>

class ccIO8501 : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8501 frame grabber or a Cognex MVS-8511 (and MVS-8511e) frame grabber when used with cable 300-0390 and the pass-through TTL connection module, P/N 800-5818-1.

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams

The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines();
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines();
```

Returns the number of output lines for this hardware configuration.

ccIO8504

```
#include <ch_cvl/pioconfg.h>

class ccIO8504 : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8504 (and MVS-8504e) frame grabber or a Cognex MVS-8514 (and MVS-8514e) frame grabber when used with cable 300-0390 and the pass-through TTL connection module.

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine

```
virtual void isValidInputLine(c_UInt8 linenumber) const;
```

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams

The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines();
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const;
```

Returns the number of output lines for this hardware configuration.

ccIO8600DualLVDS

```
#include <ch_cvl/pioconfg.h>

class ccIO8600DualLVDS : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on Cognex MVS-8600 and MVS-8600e frame grabbers when used with cable 300-0538 and the 800-5885-1 I/O connection module. The MVS-8600 and MVS-8600e series support cameras that use the Camera Link LVDS protocol.

Use this class to support I/O in the case shown in Table 1.

Model	Camera connection type	Code
MVS-8601 and MVS-8601e	Not used	
MVS-8602 and MVS-8602e	Two line scan cameras, both using LVDS encoders	2LL

Table 1. Supported I/O cases for ccIO8600DualLVDS

Wires are connected from strobe, trigger, and encoder devices according to the pinout information in Table 2. Codes in Table 2 have the following meanings:

- S*n* = Strobe for camera port *n*
- T*n* = Trigger for camera port *n*
- A*n*, B*n* = A and B channel connections from LVDS encoder for camera port *n*
- The camera connection type code is from Table 1

MVS-8602 and MVS-8602e only	Connector pins		
2LL	**Hirose (P3)**	**P4**	**P6**
A0	3	11	
B0	5	22	
A1	12		11
B1	8		22
A0’	4	13	
B0’	6	24	
A1’	7		13
B1’	9		24
T0+, T1+		5	
T0–		7	
T1–		9	
S1+		1	
S1–		3	
S0+		4	
S0–		6	

Table 2. Wire connection cases for ccIO8600DualLVDS

For this I/O configuration, trigger and strobe connections are opto-isolated.

Some pin connections are duplicated between the Hirose connector on the back faceplate of the MVS-8600 and MVS-8600e, and the onboard 26-pin headers labeled P4 and P6.

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines() const;`

Returns the number of output lines for this hardware configuration.

NOTES

ccIO8600LVDS

```
#include <ch_cvl/pioconfg.h>

class ccIO8600LVDS : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on Cognex MVS-8600 and MVS-8600e frame grabbers when used with cable 300-0539 and the 800-5885-1 I/O connection module. The MVS-8600 and MVS-8600e series support cameras that use the Camera Link LVDS protocol. This class is the default I/O class for an instance of **cc8600**.

Use this class to support I/O in the cases shown in Table 1.

Model	Camera connection type	Code
MVS-8601, MVS-8601e, MVS-8602, MVS-8602e	One area scan camera	1A
	One line scan camera using an LVDS encoder	1LL
MVS-8602 and MVS-8602e only	Two area scan cameras	2A
	One line scan camera using an LVDS encoder, plus one area scan camera	1LL+1A

Table 1.Supported I/O cases for ccIO8600LVDS

When connecting only one line scan camera to the MVS-8602 or MVS-8602e, it must be connected to camera port 0. When connecting one line scan and one area scan camera, the line scan camera must be connected to camera port 0.

Wires are connected from strobe, trigger, and encoder devices according to the pinout information in Table 2. Codes in Table 2 have the following meanings:

- S*n* = Strobe for camera port *n*
- T*n* = Trigger for camera port *n*
- A0, B0 = A and B channel connections from LVDS encoder for camera port 0
- The camera connection type codes are from Table 1

All Models		MVS-8602 and MVS-8602e only		Connector pins		
1A	**1LL**	**2A**	**1LL+1A**	**Hirose (P3)**	**P4**	**P6**
	A0		A0	3	11	
	B0		B0	5	22	
S0	S0	S0	S0	12		11
T0	T0	T0	T0	8		22
	A0’		A0’	4	13	
	B0’		B0’	6	24	
		S1	S1	7		13
		T1	T1	9		24

Table 2. Wire connection cases for ccIO8600LVDS

Some pin connections are duplicated between the Hirose connector on the back faceplate of the MVS-8600 and MVS-8600e, and the onboard 26-pin headers labeled P4 and P6.

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

ccParallelIO::BadParams

The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines();
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const;
```

Returns the number of output lines for this hardware configuration.

NOTES

ccIO8600TTL

```
#include <ch_cvl/pioconfg.h>

class ccIO8600TTL : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on Cognex MVS-8600 MVS-8600e frame grabbers when used with cable 300-0540 and the 800-5885-1 I/O connection module. The MVS-8600 and MVS-8600e series support cameras that use the Camera Link LVDS protocol.

Use this class to support I/O in the cases shown in Table 1.

Model	Camera connection type	Code
MVS-8601, MVS-8601e MVS-8602, MVS-8602e	One area scan camera	1A
	One line scan camera using a TTL encoder	1LT
MVS-8602 and MVS-8602e only	Two area scan cameras	2A
	Two line scan cameras, both using TTL encoders	2LT
	One line scan camera using a TTL encoder, plus one area scan camera	1LT+1A

Table 1.Supported I/O cases for ccIO8600TTL

When connecting only one line scan camera to the MVS-8602 or MVS-8602e, it must be connected to camera port 0. When connecting one line scan and one area scan camera, the line scan camera must be connected to camera port 0.

Wires are connected from strobe, trigger, and encoder devices according to the pinout information in Table 2. Codes in Table 2 have the following meanings:

- S*n* = Strobe for camera port *n*
- T*n* = Trigger for camera port *n*
- A*n,* B*n* = A and B channel connections from TTL encoder for camera port *n*
- The camera connection type codes are from Table 1

All Models		MVS-8602 and MVS-8602e only			Connector pins		
1LT	**1A**	**2A**	**2LT**	**1LT+1A**	**Hirose (P3)**	**P4**	**P6**
A0			A0	A0	3	11	
B0			B0	B0	5	22	
S0	S0	S0	S0	S0	12		11
T0	T0	T0	T0	T0	8		22
			A1		4	13	
			B1		6	24	
		S1	S1	S1	7		13
		T1	T1	T1	9		24

Table 2. Wire connection cases for ccIO8600TTL

Some pin connections are duplicated between the Hirose connector on the back faceplate of the MVS-8600 and MVS-8600e, and the onboard 26-pin headers labeled P4 and P6.

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s

The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines();
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const;
```

Returns the number of output lines for this hardware configuration.

NOTES

ccIOConfig

```
#include <ch_cvl/pioconfg.h>

class ccIOConfig;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	No

ccIOConfig is an abstract base used to check the validity of logical line numbers for an installed hardware I/O configuration. The derived classes are used to specify the I/O configuration of your frame grabber.

Derived class	Description
ccIOStandardOption	Specifies: • Universal parallel I/O board in standard configuration, P/N 800-5726-1 • One or both of the ISA-based parallel I/O boards, the TTL and OPTO/TTL boards.
ccIOLightControlOption	Specifies the universal parallel I/O board in light control configuration, P/N 800-5726-2
ccIOExternalOption	Specifies the universal parallel I/O board in external configuration, P/N 800-5726-3, used with the Cognex external I/O module, P/N 800-5712-2
ccIO8501 **ccIO8504**	Specifies the sixteen bidirectional TTL lines on an MVS-8500 series frame grabber, when connected to the pass-through TTL connection module (P/N 800-5818-1 with cable 300-0390.
ccIOExternal8501 **ccIOExternal8504**	Specifies the sixteen bidirectional TTL lines on an MVS-8500 series frame grabber, when connected to the external opto-isolated I/O module (P/N 800-5712-2) with cable 300-0389.

Derived class	Description
ccIOSplit8501 **ccIOSplit8504**	Specifies the sixteen bidirectional TTL lines on an MVS-8500 series frame grabber, when connected to the half TTL, half opto cable 300-0399. One branch of this cable terminates in a 10-pin screw terminal block for TTL lines. The other branch connects to the external opto-isolated I/O module, (P/N 800-5712-2).
ccIO8600LVDS	Specifies the I/O configuration appropriate when connecting one or two area scan cameras, or one line scan camera on camera port 0 using an LVDS encoder, with or without one area scan camera. This is the default I/O configuration for the MVS-8600. Use this configuration with cable 300-0539 and the 800-5885-1 I/O connection module.
ccIO8600TTL	Specifies the I/O configuration appropriate when connecting one or two line scan cameras using TTL encoders, one or two area scan cameras, or one line scan camera with TTL encoder plus one area scan camera. Use this configuration with cable 300-0540 and the 800-5885-1 I/O connection module.
ccIO8600DualLVDS	Specifies the I/O configuration appropriate when connecting two line scan cameras both using LVDS encoders. Use this configuration with cable 300-0538 and the 800-5885-1 I/O connection module.

To specify one of these I/O configurations, use the **setIOConfig()** member function of the class that represents your frame grabber. For example, **cc8501::setIOConfig()**.

Constructors/Destructors

This is an abstract base class.

Public Member Functions

isValidInputLine

```
virtual void isValidInputLine(c_UInt8 linenumber)
  const = 0;
```

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

ccParallelIO::BadParams
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber)
  const = 0;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines() const = 0;
```

Returns the total number of usable input lines for this I/O configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const = 0;
```

Returns the total number of usable output lines for this I/O configuration.

ccIOConfig

ccIOExternal8500l

```
#include <ch_cvl/pioconfg.h>

class ccIOExternal8500l : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8500L frame grabber when used with cable 300-0389 and the external opto-isolated I/O module, P/N 800-5712-2.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

ccIOExternal8500I

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines();`

Returns the number of output lines for this hardware configuration.

ccIOExternal8501

```
#include <ch_cvl/pioconfg.h>

class ccIOExternal8501 : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8501 frame grabber or a Cognex MVS-8511 (and MVS-8511e) frame grabber when used with cable 300-0389 and the external opto-isolated I/O module, P/N 800-5712-2.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

ccIOExternal8501

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines();`

Returns the number of output lines for this hardware configuration.

ccIOExternal8504

```
#include <ch_cvl/pioconfg.h>

class ccIOExternal8504 : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8504 (and MVS-8504e) frame grabber or a Cognex MVS-8514 (and MVS-8514e) frame grabber when used with cable 300-0389 and the external opto-isolated I/O module, P/N 800-5712-2.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

ccIOExternal8504

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines();`

Returns the number of output lines for this hardware configuration.

ccIOExternalOption

```
#include <ch_cvl/pioconfg.h>

class ccIOExternalOption : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers for a parallel I/O board in external configuration (Cognex P/N 800-5726-3), which is used with the Cognex external I/O module (P/N 800-5712-2).

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine

```
virtual void isValidInputLine(c_UInt8 linenumber) const;
```

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number

Throws

ccParallelIO::BadParams

The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines() const;
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const;
```

Returns the number of output lines for this hardware configuration.

ccIOLightControlOption

```
#include <ch_cvl/pioconfg.h>

class ccIOLightControlOption : public ccIOLightControl8100;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers for a parallel I/O board in light control configuration (Cognex P/N 800-5726-2).

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine

```
virtual void isValidInputLine(c_UInt8 linenumber) const;
```

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number

Throws

ccParallelIO::BadParams

The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines() const;
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const;
```

Returns the number of output lines for this hardware configuration.

ccIOSplit8500l

```
#include <ch_cvl/pioconfg.h>

class ccIOSplit8500l : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8500L frame grabber when used with cable 300-0399. One branch of this cable connects to the external opto-isolated I/O module, P/N 800-5712-2; the other branch terminates in a Phoenix screw terminal strip.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

ccIOSplit8500I

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines();`

Returns the number of output lines for this hardware configuration.

ccIOSplit8501

```
#include <ch_cvl/pioconfg.h>

class ccIOSplit8501 : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8501 frame grabber or a Cognex MVS-8511 (and MVS-8511e) frame grabber when used with cable 300-0399. One branch of this cable connects to the external opto-isolated I/O module, P/N 800-5712-2; the other branch terminates in a Phoenix screw terminal strip.

Public Member Functions

isValidInputLine `virtual void isValidInputLine(c_UInt8 linenumber) const;`

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

`virtual void isValidOutputLine(c_UInt8 linenumber) const;`

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines();`

Returns the number of output lines for this hardware configuration.

ccIOSplit8504

```
#include <ch_cvl/pioconfg.h>

class ccIOSplit8504 : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers available on a Cognex MVS-8504 (and MVS-8504e) frame grabber or a Cognex MVS-8514 (and MVS-8514e) frame grabber when used with cable 300-0399. One branch of this cable connects to the external opto-isolated I/O module, P/N 800-5712-2; the other branch terminates in a Phoenix screw terminal strip.

Public Member Functions

isValidInputLine

```
virtual void isValidInputLine(c_UInt8 linenumber) const;
```

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams
The specified output line number is not valid for this I/O configuration.

numInputLines `virtual c_UInt32 numInputLines();`

Returns the number of input lines for this hardware configuration.

numOutputLines `virtual c_UInt32 numOutputLines();`

Returns the number of output lines for this hardware configuration.

ccIOStandardOption

```
#include <ch_cvl/pioconfg.h>

class ccIOStandardOption : public ccIOConfig;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class is used to check the validity of I/O logical line numbers for a parallel I/O board in standard configuration (Cognex P/N 800-5726-1).

Constructors/Destructors

A default-constructed instance of this class is intended to be used only as an argument to **setIOConfig()**.

Public Member Functions

isValidInputLine

```
virtual void isValidInputLine(c_UInt8 linenumber) const;
```

No effect if specified input line number is valid.

Parameters

linenumber The input line number.

Throws

*ccParallelIO::BadParam*s
The specified input line number is not valid for this I/O configuration.

isValidOutputLine

```
virtual void isValidOutputLine(c_UInt8 linenumber) const;
```

No effect if specified output line number is valid.

Parameters

linenumber The output line number.

Throws

ccParallelIO::BadParams

The specified output line number is not valid for this I/O configuration.

numInputLines

```
virtual c_UInt32 numInputLines() const;
```

Returns the number of input lines for this hardware configuration.

numOutputLines

```
virtual c_UInt32 numOutputLines() const;
```

Returns the number of output lines for this hardware configuration.

ccKeyboardEvent

```
#include <ch_cvl/uievent.h>

class ccKeyboardEvent;
```

Class Properties

Copyable	No
Derivable	Not intended
Archiveable	Complex

This class encapsulates information associated with a keyboard event. For example, when a keyboard key is pressed down.

Constructors/Destructors

ccKeyboardEvent

```
ccKeyboardEvent ();
```

Enumerations

Event

This enumeration specifies types of keyboard events.

Value	Meaning
eNone	There is no key associated with this mouse event.
eKeyDown	A keyboard key was depressed. (Also, see **repeatKey()**).
eKeyUp	A keyboard key was released.
eKeyPress	An ASCII or Unicode key was pressed down.

eKeyDown and *eKeyUp* events are only generated for virtual keys.

ccKeyboardEvent

KeyType This enumeration specifies key types for keyboard events.

Value	Meaning
eAscii	An ASCII key.
eUnicode	A Unicode key.
eVirtual	A virtual key

VirtualKey This enumeration specifies virtual key types associated with keyboard events.

Value	Meaning
eNoVKey = 0	The keyboard event was not caused by a virtual key.
eBackspace = 0x08	**Backspace** key.
eTab	**Tab** key.
eReturn = 0xD0	**Enter** key.
eShift = 0x10	**Shift** key.
eControl	**Ctrl** key.
eMenu	Menu key.
ePause	**Pause** key.
eCapLock	**Caps Lock** key.
eEscape = 0x1B	**Esc** key.
eSpace = 0x20	Space bar.
ePrior	**Page Up** key.
eNext	**Page Down** key.
eEnd	**End** key.
eHome	**Home** key.
eLeft	Left Arrow key.
eUp	Up Arrow key.
eRight	Right Arrow key.
eDown	Down Arrow key.

Value	Meaning
eInsert = 0x2D	**Insert** Key
eDelete	**Delete** key.
eHelp	**F1** function key.
eF1 = 0x70	**F1** function key.
eF2	**F2** function key.
eF3	**F3** function key.
eF4	**F4** function key.
eF5	**F5** function key.
eF6	**F6** function key.
eF7	**F7** function key.
eF8	**F8** function key.
eF9	**F9** function key.
eF10	**F10** function key.
eF11	**F11** function key.
eF12	**F12** function key.
eF13	**F13** function key.
eF14	**F14** function key.
eF15	**F15** function key.
eF16	**F16** function key.
eF17	**F17** function key.
eF18	**F18** function key.
eF19	**F19** function key.
eF20	**F20** function key.
eF21	**F21** function key.
eF22	**F22** function key.

Value	Meaning
eF23	**F23** function key.
eF24	**F24** function key.

Public Member Functions

event

```
void event(Event e);
Event event() const;
```

- `void event(Event e);`

  Sets the keyboard event type. Must be one of the **Event** enums.

  Parameters

  e The event type.

- `Event event() const;`

  Returns the keyboard event type.

key

```
void key(char ch);
void key(wchar_t uc);
void key(VirtualKey vk);
```

- `void key(char ch);`

  Sets the key for this event (an ASCII character).

  Parameters

  ch The ASCII character.

- `void key(wchar_t uc);`

  Sets the key for this event (a Unicode character).

  Parameters

  uc The Unicode character.

- `void key(VirtualKey vk);`

  Sets the key for this event (a virtual key). Must be one of the **VirtualKey** enums.

  Parameters

  vk The virtual key character.

keyType `KeyType keyType() const;`

Returns the key type; ASCII, Unicode, or virtual.

Use this method to determine which of the following three methods to call to get the key code.

asciiKey `TCHAR asciiKey() const;`

Returns an ASCII key character.

unicodeKey `c_Int16 unicodeKey() const;`

Returns a Unicode key character.

virtualKey `VirtualKey virtualKey() const;`

Returns a virtual key.

repeatKey

```
void repeatKey(bool b);
bool repeatKey() const;
```

Specifies whether an **Event::eKeyDown** event is due to a key press or a key repeat. If set true, its due to the repeat key. If set false, its due to a key press.

- `void repeatKey(bool b);`

  Set the repeatKey flag, true or false.

  Parameters

  b The new flag value.

- `bool repeatKey() const;`

  Return the repeatKey flag.

Static Functions

toVirtualKey

```
static VirtualKey toVirtualKey(c_Int32 key);
```

Returns the virtual key corresponding to the specified value. Returns *eNoKey* if the value is not valid.

Parameters

key	The value to convert to a virtual key.

ccLabeledProjection

```
#include <ch_cvl/lablproj.h>

class ccLabeledProjection;
```

A name space class that holds enumerations and constants used with labeled projection.

Enumerations

BinOrder

```
enum BinOrder
```

This enumeration defines the bin ordering methods used to construct circular and radial projection models.

Value	Meaning
eRadial	Bins are numbered from innermost to outermost annulus within each sector.
eAngular	Bins are numbered from first to last sector within each annulus.
kDefaultBinOrder	The default bin order, as defined in *lablproj.h*

Orientation

```
enum Orientation
```

This enumeration defines the direction in which angles are measured and the order in which sectors are numbered.

Value	Meaning
eClockwise	Clockwise
eCounterClockwise	Counter-clockwise
kDefaultOrientation	The default orientation, as defined in *lablproj.h*

ccLabeledProjection

ccLabeledProjectionModel

```
#include <ch_cvl/lablproj.h>

template <class M>
class ccLabeledProjectionModel;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class represents a projection model used by the Labeled Projection tool. A projection model consists of an image where the value of each pixel in the image determines the bin into which the value of the corresponding pixel in an input image is added.

This class is a templated class, where

M — Template parameter specifying the value associated with each pixel in the model. CVL provides instantiations of **ccLabeledProjectionModel** for the types **c_UInt8**, **c_UInt16**, and **c_UInt32**

Constructors/Destructors

ccLabeledProjectionModel

```
ccLabeledProjectionModel();
```

A **ccLabeledProjectionModel** created by the default constructor cannot be used to project an image.

Public Member Functions

train

```
void train(const ccPelBuffer_const<M> &model);
```

Trains this **ccLabeledProjectionModel** using the supplied image as the projection model.

Parameters

model — The image to use as the projection model. The template type for *model* must be an integer.

Throws

ccLabeledProjection::UnboundImage
model is unbound.

ccLabeledProjection::BadImage
Every pixel in *model* has a value of 0.

Notes

Any previous training information is discarded.

hist

```
const cmStd vector<c_UInt32>& hist() const;
```

Returns a histogram of the distribution of pixel values in this **ccLabeledProjectionModel**. This histogram tells you how many pixels in the model are associated with each bin in the projection image produced using the model.

histInv

```
const cmStd vector<float>& histInv() const;
```

Returns the inverse of the histogram of the distribution of pixel values in this **ccLabeledProjectionModel**. Each element of the returned vector of **float**s is equal to the inverse of the corresponding element of the vector of **c_UInt32**s returned by **hist()**.

model

```
const ccPelBuffer_const<M>& model() const;
```

Returns a **ccPelBuffer_const<M>** containing the projection model.

offset

```
ccIPair offset() const;
void offset(ccIPair offset);
```

- `ccIPair offset() const;`

  Returns the image offset of this **ccLabeledProjectionModel**'s projection model.

- `void offset(ccIPair offset);`

  Sets the image offset of this **ccLabeledProjectionModel**'s projection model.

  Parameters

  offset The image offset to set.

projectionLength

```
c_Int32 projectionLength() const;
```

Returns the projection length of this **ccLabeledProjectionModel**. The projection length is equal to the largest value found in this **ccLabeledProjectionModel**'s projection model. The projection length is the minimum size that must be allocated for a projection image created using this **ccLabeledProjectionModel**.

maxPel

```
c_Int32 maxPel() const;
```

Returns the same value as **projectionLength()**.

isTrained

```
bool isTrained() const;
```

Returns false if this **ccLabeledProjectionModel** is default-constructed, true otherwise.

ccLabeledProjectionModel

ccLine

```
#include <ch_cvl/shapes.h>

class ccLine : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a line passing through a point in a given direction. The direction can be specified as a vector or as an angle. Lines have infinite length. To specify line segments of finite length, see **ccLineSeg**.

Constructors/Destructors

ccLine

```
ccLine();

ccLine(const cc2Vect& dir, const cc2Vect& pos);

ccLine(const ccRadian& t, const cc2Vect& pos);

ccLine(const cc2Vect& v);
```

- `ccLine();`

  Default constructor. Constructs a line with direction (1,0), that is horizontal, passing through the point (0,0).

- `ccLine(const cc2Vect& dir, const cc2Vect& pos);`

  Constructs a line in the direction *dir* passing through the point *pos*.

  Parameters

dir	The direction of the line. Must be a unit vector.
pos	The point through which the line passes.

  Notes

  If *dir* is not a unit (normalized) vector, then projections, distances, and other mathematical line operations may be calculated incorrectly.

Throws

ccShapesError::DegenerateShape
dir is (0,0).

- `ccLine(const ccRadian& t, const cc2Vect& pos);`

Constructs a line with direction *t* passing through the point *pos*.

Parameters

t	The direction of the line in radians.
pos	The point through which the line passes.

- `ccLine(const cc2Vect& v);`

Conversion constructor. Constructs a line in the direction **v.angle()** passing through the point (**v.x()**, **v.y()**). Essentially, this is a line passing through the points (0,0) and *v*.

Parameters

v	The vector the specifies the angle and the point through which the line passes.

Throws

ccMathError::NullVector
v is (0,0).

Operators

operator== `bool operator==(const ccLine& line) const;`

Returns true if this line has the same direction and position as another line.

Parameters

line	The other line.

operator!= `bool operator!=(const ccLine& line) const;`

Returns true if this line is not equal to another line.

Parameters

line	The other line.

Public Member Functions

dir

```
const cc2Vect& dir() const;
cc2Vect& dir();
```

- `const cc2Vect& dir() const;`

  Gets the direction vector for this line.

- `cc2Vect& dir();`

  This overload is deprecated. Use the previous overload, which returns a **const** vector, instead.

pos

```
const cc2Vect& pos() const;
cc2Vect& pos();
void pos(const cc2Vect& p);
```

- `const cc2Vect& pos() const;`

  Gets the point through which this line is constrained to pass.

  Notes

  For a line created with the conversion constructor **ccLine(const cc2Vect& v)**, this function returns the point (**v.x()**, **v.y()**) and not (0,0).

- `cc2Vect& pos();`

  This overload is deprecated. Use the previous overload, which returns a **const** vector, instead.

- `void pos(const cc2Vect& p);`

  Sets the point through which this line is constrained to pass.

  Parameters

p	The point.

map

```
ccLine map(const cc2Xform& c) const;
void map(const cc2Xform& c, ccLine& result) const;
```

- `ccLine map(const cc2Xform& c) const;`

  Returns a line that is the result of mapping this line with the transformation object *c*.

  Parameters

c	The transformation object.

- `void map(const cc2Xform& c, ccLine& result) const;`

  Sets *result* to be a line that is the result of mapping this line with the transformation object *c*.

  Parameters

c	The transformation object.
result	The mapped line.

angle

```
ccRadian angle() const;
void angle(ccRadian a);
ccRadian angle (const ccLine& l) const;
```

- `ccRadian angle() const;`

  Returns the line angle. The returned angle is greater than $-\pi$ and less than or equal to $+\pi$.

- `void angle(ccRadian a);`

  Sets the line angle.

  Parameters

a	The angle in radians.

- `ccRadian angle (const ccLine& l) const;`

  Returns the angle from this line to the given *line*. The returned angle is greater than $-\pi$ and less than or equal to $+\pi$.

  Parameters

l	The other line.

intersect

```
ccPoint intersect(const ccLine& line) const;
```

Returns the point at which this line intersects with the given *line*.

Parameters

line The other line.

Throws

ccShapesError::LinesParallel
The two lines are parallel.

parallel

```
ccLine parallel(const cc2Vect& point) const;
```

Returns a line parallel to this one that passes through the given *point*.

Parameters

point The point through which the parallel line passes.

isParallel

```
bool isParallel(const ccLine& line) const;
```

Returns true if this line is parallel to the given *line*.

Parameters

line The other line.

normal

```
ccLine normal(const cc2Vect& point) const;
```

Returns the line rotated 90 degrees through the given *point*.

Parameters

point The point through which the rotated line passes.

proj

```
ccPoint proj(const cc2Vect& point) const;
```

Returns the projection of the given *point* onto this line. In other words, returns the point *p* such that the line **normal(p)** passes through the given *point*.

Parameters

point The point to project.

offset

```
double offset(const cc2Vect& point) const;
```

Returns the signed distance from **pos()** to the projection of the point on this line.

Parameters

point The point to project.

clone `virtual ccShapePtrh clone() const;`

Returns a pointer to a copy of this directional line.

isOpenContour `virtual bool isOpenContour() const;`

Returns true if this shape is an open contour. For directional lines, this function always returns false.

See **ccShape::isOpenContour()** for more information.

isRegion `virtual bool isRegion() const;`

Returns true if this shape is a region. For directional lines, this function always returns false.

See **ccShape::isRegion()** for more information.

isFinite `virtual bool isFinite() const;`

Returns true if this shape is finite. For directional lines, this function always returns false.

See **ccShape::isFinite()** for more information.

isEmpty `virtual bool isEmpty() const;`

Returns true if the set of points that lie on the boundary of this shape is empty. For directional lines, this function always returns false.

See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

Returns true if the set of points on the boundary of this shape is infinite, and there is a well-defined tangent at all but a finite number of these points. For directional lines, this function always returns true.

See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

Returns true if this shape is already decomposed. For directional lines, this function always returns false.

See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

Returns true if this shape can be reversed. For directional lines, this function always returns true.

Notes

ccLines are reversible, but **ccFLine**s are not.

See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Throws

ccShapesError::InfiniteExtent
For all **ccLine**s.

See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on this directional line to the given point.

Parameters

p The point.

See **ccShape::nearestPoint()** for more information.

perimeter

```
virtual double perimeter() const;
```

Throws

ccShapesError::InfiniteExtent
For all **ccLine**s.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this line to the given point, as determined by **nearestPoint()**.

Parameters

info Shape information for this line.

point The point.

See **ccShape::nearestPerimPos()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this directional line.

See **ccShape::reverse()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Throws

ccShapesError::InfiniteExtent
For all **ccLine**s.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this directional line mapped by *X*.

Parameters

X	The transformation object.

Notes

If *X* is singular, the mapping may collapse this line to a single point.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccGeneralShapeTree** without any children.

See **ccShape::decompose()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the line segment describing the portion of this line over the given range. The length of the final returned line segment is equal to the absolute value of the distance component of *range*.

Parameters

info	Shape information for this line.
range	The perimeter range.

See **ccShape::subShape()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

ccLineFitDefs

```
#include <ch_cvl/fit.h>

class ccLineFitDefs;
```

A name space that holds enumerations and constants used with the Line Fitting tool (**cfLineFit()**).

Enumerations

fit_mode

```
enum fit_mode
```

This enumeration defines types of fitting supported by the Line Fitting tool.

Value	Meaning
eLeastSquares	Fits a line by minimizing the sum of the squares of the distances between the points and the fit line.
eMinMaxDistance	Fits a line by minimizing the maximum distance between any one point and the fit line.
kDefaultFitMode = eLeastSquares	

ccLineFitDefs

ccLineFitParams

```
#include <ch_cvl/fit.h>

class ccLineFitParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds the parameters for the Line Fitting tool. See **cfLineFit()**.

Constructors/Destructors

ccLineFitParams

```
ccLineFitParams(
  enum ccLineFitParams::fit_mode fitMode =
                          ccLineFitParams::kDefaultFitMode,
  c_UInt32 numIgnore=0,
  double threshold=HUGE_VAL);
```

Constructs a **ccLineFitParams** params object

Parameters

fitMode — The type of fitting to perform. *fitMode* must be one of the following values:

ccLineFitParams::eLeastSquares
ccLineFitParams::eMinMaxDistance

numIgnore — The number of outlying points to ignore.

threshold — The maximum error for a valid fit. If the fitted line has an error equal to or greater than *threshold*, it is not considered a valid fit.

Throws

ccLineFitParams::BadParams
threshold is less than 0.

Operators

operator==

```
bool operator== (const ccLineFitParams& that) const;
```

Two **ccLineFitParams** objects are considered equal if and only if the values returned by **fitMode()**, **numIgnore()**, and **threshold()** are equal.

Public Member Functions

fitMode

```
enum ccLineFitParams::fit_mode fitMode() const;
```

```
void fitMode(enum ccLineFitParams::fit_mode fitMode);
```

- ```
  enum ccLineFitParams::fit_mode fitMode() const;
  ```

  Returns the type of fitting this **ccLineFitParams** is configured to perform. The function returns one of the following values:

  ccLineFitParams::eLeastSquares
  ccLineFitParams::eMinMaxDistance

- ```
  void fitMode(enum ccLineFitParams::fit_mode fitMode);
  ```

  Sets the type of fitting this **ccLineFitParams** is configured to perform.

  Parameters

  fitMode — The type of fitting to perform. *fitMode* must be one of the following values:

  ccLineFitParams::eLeastSquares
  ccLineFitParams::eMinMaxDistance

numIgnore

```
c_UInt32 numIgnore() const;
```

```
void numIgnore(c_UInt32 numIgnore);
```

- ```
  c_UInt32 numIgnore() const;
  ```

  Returns the number of outlying points ignored by this **ccLineFitParams**.

- ```
  void numIgnore(c_UInt32 numIgnore);
  ```

  Sets the number of outlying points ignored by this **ccLineFitParams**.

Parameters

numIgnore The number of points to ignore.

---

threshold

```
double threshold() const;
void threshold(double threshold);
```

- `double threshold() const;`

  Returns the current threshold, which is checked by the Line Fitting tool as it evaluates potential fit lines. If the error value calculated by the Line Fitting tool equals or exceed this value, the potential fit line is not valid.

- `void threshold(double threshold);`

  Sets the threshold. If the error value calculated by the Line Fitting tool for a potential fit line is greater than or equal to this value, the line is invalid.

  Parameters

  threshold The error threshold. *threshold* can have any positive value.

  If the fit mode is *ccLineFitParams::eLeastSquares*, the threshold is compared to the computed least squares value. If the fit mode is *ccLineFitParams::eMinMaxDistance*, the threshold is compared to the maximum distance between a point and the line.

  Throws

  ccLineFitParams::BadParams
  threshold is less than 0.

ccLineFitParams

ccLineFitResults

```
#include <ch_cvl/fit.h>

class ccLineFitResults;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that holds the results from the Line Fitting tool. See **cfLineFit()**.

Constructors/Destructors

ccLineFitResults

```
ccLineFitResults();
```

Constructs a **ccLineFitResults** with no result.
found() = false, **error()** = 0, **time()** = 0.

Operators

operator==

```
bool operator== (const ccLineFitResults& that) const;
```

Two **ccLineFitResults** objects are considered equal if and only if the values returned by **found**, **line**, and **error** are equal.

Public Member Functions

found

```
bool found() const;
```

Returns true if the RMS error of this line is less than the threshold you supplied. Returns false otherwise.

reset

```
void reset();
```

Reset this **ccLineFitResults**. Calling **found** returns false after calling this function.

ccLineFitResults

line

```
const ccFLine &line() const;
```

Returns the fitted line.

error

```
double error() const;
```

The measured error between the data and the estimated line. The default value is 0.

Notes

The way the error is computed is specified by the fit mode. For *ccLineFitDefs::eLeastSquares*, the error is the root mean squared error between the selected subset of points and the fitted line. For *ccLineFitDefs::eMinMaxDistance*, the error is the maximum distance between the selected subset of points and the fitted line.

runParams

```
const ccLineFitParams &runParams() const;
```

Returns the **ccLineFitParams** used to fit this line. If **found** returns false, then this function's return value is undefined.

outliers

```
const cmStd vector<c_UInt32> &outliers() const;
```

Returns the indices of the points that were ignored. If **found** returns false, then this function's return value is undefined.

time

```
double time() const;
```

Returns the time in milliseconds required to compute this result. If **found** returns false, then this function's return value is undefined.

ccLineScanDistortionCorrection

```
#include <ch_cvl/lnscdist.h>

class ccLineScanDistortionCorrection;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccLineScanDistortionCorrection** class contains all the methods necessary to train and run the Line Scan Distortion Correction tool.

Constructors/Destructors

ccLineScanDistortionCorrection

```
ccLineScanDistortionCorrection();

~ccLineScanDistortionCorrection();

ccLineScanDistortionCorrection(const
  ccLineScanDistortionCorrection&);
```

- `ccLineScanDistortionCorrection();`

  Constructs a default-constructed line scan distortion object.

- `~ccLineScanDistortionCorrection();`

  Destructor.

- `ccLineScanDistortionCorrection(const ccLineScanDistortionCorrection&);`

  Copy constructor.

  Parameters

  ccLineScanDistortionCorrection
  The source of the copy.

Operators

```
ccLineScanDistortionCorrection& operator= (const
 ccLineScanDistortionCorrection&);
```

Assignment operator.

Parameters

ccLineScanDistortionCorrection
The assignment source.

operator==

```
bool operator==(const ccLineScanDistortionCorrection
 &that) const;
```

Returns true if *this* object equals to *that* object. Returns false otherwise.

Parameters

that The line scan distortion object to compare with this line scan distortion object.

Notes

All members must be equal to double precision in order to be considered equal.

Public Member Functions

isTrained

```
bool isTrained() const;
```

Returns whether or not this distortion tool has been trained.

train

```
void train(const cmStd vector<double>
 &positionsOfRegularPeriodicPattern,
 const ccPelSpan &xExtents);

void train(const ccPelBuffer_const<c_UInt8> &trainImage);
```

- ```
  void train(const cmStd vector<double>
   &positionsOfRegularPeriodicPattern,
   const ccPelSpan &xExtents);
  ```

  Manually trains the Line Scan Distortion Correction tool from positions of the regular period pattern.

  Parameters

  positionsOfRegularPeriodicPattern
  The positions of the regular period pattern.
  ```

xExtents The x extents for the tool.

Notes

Retrains the tool even if it was already trained

Throws

ccLineScanDistortionCorrectionDefs::BadParams
The positions are not monotonically increasing.

ccLineScanDistortionCorrectionDefs::BadParams
Any of the positions fall outside the specified range.

- `void train(const ccPelBuffer_const<c_UInt8> &trainImage);`

Manually trains the Line Scan Distortion Correction tool from positions of the regular period pattern.

Parameters

trainImage The supplied training image.

Notes

Retrains the tool even if it was already trained.

Throws

ccPel::UnboundWindow
The image is unbound.

ccLineScanDistortionCorrectionDefs::BadImage
The tool cannot find at least four instances of the repeating pattern in the training image.

positionsOfRegularPeriodicPattern

```
cmStd vector<double> positionsOfRegularPeriodicPattern()
 const;
```

Returns the positions used to train the tool.

Throws

ccLineScanDistortionCorrectionDefs::NotTrained
This tool was not trained from a set of *positionsOfRegularPeriodicPattern*

trainImage `const ccPelBuffer_const<c_UInt8> &trainImage() const;`

Returns the positions used to train the tool.

Throws

ccLineScanDistortionCorrectionDefs::NotTrained
This tool was not trained from a set of *positionsOfRegularPeriodicPattern*

xExtents

```
const ccPelSpan &xExtents() const;
```

Returns the x extents of the trained tool.

Throws

ccLineScanDistortionCorrectionDefs::NotTrained
This tool was not trained (neither manually trained nor image-based trained).

---

run

```
void run(const ccPelBuffer_const<c_UInt8> &srcImage,
 ccPelBuffer<c_UInt8> &dstImage) const;

ccPelBuffer<c_UInt8> run(const ccPelBuffer_const<c_UInt8>
 &src) const;
```

-
```
void run(const ccPelBuffer_const<c_UInt8> &srcImage,
 ccPelBuffer<c_UInt8> &dstImage) const;
```

Undistorts the source image and stores the results in the destination image.

Parameters

srcImage The source image.

dstImage The destination image.

Notes

This function follows the same behavior patterns as CVL pelfuncs. See *EFFECTS COMMON TO ALL MULTIPLE WINDOWS FUNCTIONS* in *<ch_cvl/pelfunc.h>*

The input image's client coordinates will be copied to the output image's client coordinates.

Throws

ccLineScanDistortionCorrectionDefs::NotTrained
This tool is untrained.

ccLineScanDistortionCorrectionDefs::MismatchedRanges
The x extents of the source image do not exactly match the trained x extents.

- `ccPelBuffer<c_UInt8> run(const ccPelBuffer_const<c_UInt8> &src) const;`

  Undistorts the source image and returns the result.

  Parameters

  src The source image.

  Notes

  The output image will have the same size as the source image.

  The input image's client coordinates will be copied to the output image's client coordinates.

  Throws

  ccLineScanDistortionCorrectionDefs::NotTrained
  This tool is untrained.

  ccLineScanDistortionCorrectionDefs::MismatchedRanges
  The x extents of the source image do not exactly match the trained x extents.

ccLineScanDistortionCorrection

ccLineSeg

```
#include <ch_cvl/shapes.h>

class ccLineSeg : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a line segment of finite length passing through two points. To specify lines, see **ccLine**.

Constructors/Destructors

ccLineSeg

```
ccLineSeg();

ccLineSeg(const cc2Vect& p1, const cc2Vect& p2);
```

- `ccLineSeg();`

  Default constructor. Creates a line segment with start and end points both equal to (0,0).

- `ccLineSeg(const cc2Vect& p1, const cc2Vect& p2);`

  Creates a line segment with the specified start and end points.

  Parameters

p1	The start point of the line segment.
p2	The end point of the line segment.

Operators

operator==

```
bool operator==(const ccLineSeg& other) const;
```

Returns true if this line segment has the same start and end points as another line segment.

Parameters

other	The other line segment.

operator!=

```
bool operator!=(const ccLineSeg& other) const;
```

Returns true if this line segment is not equal to another line segment.

Parameters

other The other line segment.

Public Member Functions

p1

```
const ccPoint& p1() const;
```

```
void p1(const cc2Vect& p);
```

- ```
  const ccPoint& p1() const;
  ```

  Gets the start point of the line segment.

- ```
  void p1(const cc2Vect& p);
  ```

  Sets the start point of the line segment.

  Parameters

  p The start point.

p2

```
const ccPoint& p2() const;
```

```
void p2(const cc2Vect& p);
```

- ```
  const ccPoint& p2() const;
  ```

  Gets the end point of the line segment.

- ```
  void p2(const cc2Vect& p);
  ```

  Sets the end point of the line segment.

  Parameters

  p The end point.

map

```
ccLineSeg map(const cc2Xform& c) const;
```

Returns a line segment that is the result of mapping this line segment with the transformation object *c*.

Parameters

c The transformation object.

line

```
ccLine line() const;
```

Converts the line segment into a line.

Throws

ccShapesError::DegenerateShape
The line segment is degenerate.

degen

```
bool degen() const;
```

Returns true if this line segment is degenerate (start point and end point are equal).

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this line segment.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For line segments, this function always returns true. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For line segments, this function always returns false. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For line segments, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For line segments, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

Thus function returns true for most line segments. It returns false for line segments of zero length. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For line segments, this function always returns true. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For line segments, this function always returns true. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this line segment. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on this line segment to the given point.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

perimeter

```
virtual double perimeter() const;
```

Returns the length of this line segment.

See **ccShape::perimeter()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this line segment to the given point, as determined by **nearestPoint()**.

Parameters

info	Shape information for this line segment.
point	The point.

See **ccShape::nearestPerimPos()** for more information.

startPoint

```
virtual cc2Vect startPoint() const;
```

Returns the starting point of this line segment. See **ccShape::startPoint()** for more information.

endPoint

```
virtual cc2Vect endPoint() const;
```

Returns the ending point of this line segment. See **ccShape::endPoint()** for more information.

startAngle

```
virtual ccRadian startAngle() const;
```

Returns the starting tangent direction of this line segment.

Throws

ccShapesError::NoTangent
hasTangent() is false for this line segment.

See **ccShape::startAngle()** for more information.

endAngle

```
virtual ccRadian endAngle() const;
```

Returns the ending tangent direction of this line segment.

Throws

ccShapesError::NoTangent
hasTangent() is false for this line segment.

See **ccShape::endAngle()** for more information.

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Returns 0.0 for a line segment.

Throws

ccShapesError::NoTangent
hasTangent() is false for this line segment.

See **ccShape::tangentRotation()** for more information.

windingAngle `virtual ccRadian windingAngle(const cc2Vect &p) const;`

Returns the signed angle determined by the rays from the given point through the start and end points of this line segment. This is sometimes called a subtended angle. The result always has a magnitude less than or equal to π radians.

Parameters

p	The point.

See **ccShape::windingAngle()** for more information.

reverse `virtual ccShapePtrh reverse() const;`

Returns the reversed version of this line segment. See **ccShape::reverse()** for more information.

sample
```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly the tangent, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores line segments for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape `virtual ccShapePtrh mapShape(const cc2Xform& X) const;`

Returns this line segment mapped by *X*.

Parameters

X	The transformation object.

See **ccShape::mapShape()** for more information.

decompose `virtual ccShapePtrh decompose() const;`

Returns a clone of this line segment. As a line segment is already a decomposed shape, this method is equivalent to **clone()** for line segments. See **ccShape::decompose()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the line segment describing the portion of this line segment over the given perimeter range. The length of the final returned line segment is equal to the absolute value of the distance component of *range*.

Parameters

info	Shape information for this line segment.
range	The perimeter range.

See **ccShape::subShape()** for more information.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

ccLiveDisplayProps

```
#include <ch_cvl/liveprop.h>

class ccLiveDisplayProps;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class allows you to control various apects of the display of live images. To influence live display, you pass a pointer to an instance of the **ccLiveDisplayProps** class as an argument to **ccDisplay::startLiveDisplay()**.

Once live display has started, you must stop it by calling **ccDisplay::stopLiveDisplay()** before you can change live display properties. You change live display properties using setters of **ccLiveDisplayProps**. The new properties take effect the next time you call **ccDisplay::startLiveDisplay()**.

Note Some member functions of the **ccLiveDisplayProps** class have no effect on certain frame grabbers or platforms. Such cases are noted where applicable.

Constructors/Destructors

ccLiveDisplayProps

```
ccLiveDisplayProps ();
```

Default constructor. Constructs a **ccLiveDisplayProps** object with the following default values:

Property	Description	Default Value
frameRateInterval	The sampling time interval (in seconds) used to compute the live display frame rate.	0
displayOutput	True causes all acquired images to be displayed. False causes all acquired images to be discarded and not displayed.	True

Property	Description	Default Value
restartDelay	The amount of delay (in seconds) inserted between the start of an acquisition and the call to complete the acquisition.	0.0 (no delay)
makeLocal	Make the pel buffer returned from a completed acquisition local as opposed to remote. This property is currently supported only on the MVS-8120 frame grabber. Using remote pel buffers can dramatically improve the performance of live display on the MVS-8120 (see also the **makeLocal()** method). This property is supported only when the magnification of the display is set to 1.0.	False
threadPriority	The priority of the live display thread. Takes effect on the first call to **startLiveDisplay()**. Subsequent calls have no effect until live display is stopped.	

Operators

operator==()

```
bool operator==(const ccLiveDisplayProps& p) const;
```

Returns true if this **ccLiveDisplayProps** object is equal to the supplied one, otherwise false.

Parameters

p The other **ccLiveDisplayProps** object.

operator!=

```
bool operator!=(const ccLiveDisplayProps& p) const;
```

Returns true if this **ccLiveDisplayProps** object is not equal to the supplied one, otherwise false.

Parameters

p The other **ccLiveDisplayProps** object.

Public Member Functions

clientTransform

```
const cc2Xform& clientTransform () const;
```

```
void clientTransform (const cc2Xform& xform);
```

- ```
  const cc2Xform& clientTransform () const;
  ```

  Returns the transform used to convert image coordinates to client coordinates.

- ```
  void clientTransform (const cc2Xform& xform);
  ```

  Sets the transform applied to each pel buffer acquired for live display. Internally calls the pel buffer's **clientFromImageXformBase()** method to set the transform.

  Parameters

  xform The transform used to convert image coordinates to client coordinates. Must not be singular.

  Notes

  Once this method has been called, the specified transform is applied to each acquired pel buffer while live display is running, overriding the pel buffer's own transform.

  Throws

  ccMathError::Singular

  xform is singular.

frameRateInterval

```
double frameRateInterval () const ;
```

```
void frameRateInterval (double time) ;
```

- ```
  double frameRateInterval () const ;
  ```

  Returns the frame rate interval.
  ```

- `void frameRateInterval (double time) ;`

  Sets the sampling time interval used to compute the live display frame rate. The default is zero. To retrieve the live display frame rate, use **ccDisplay::liveFrameRate()**.

  Parameters

  time The time in seconds. Must be greater than or equal to zero.

  Notes

  By default, **ccDisplay::liveFrameRate()** is calculated by dividing the number of frames that have been displayed by the time elapsed since **ccDisplay::startLiveDisplay()** was called. This is achieved by setting the frame rate sampling interval to zero (this is also the default).

  The live display frame rate calculation can be improved by setting the frame rate sampling interval higher than zero. This causes the calculation to be based on the most recent images within that sampling interval. The reference time of the sample is reset each time **ccDisplay::liveFrameRate()** is called.

  The live frame rate calculation is based on an exponential moving average over the sampling interval. The benefit of setting a sampling interval is that this allows the frame rate calculation to recover more quickly and to be more accurate when other external applications, or a screen saver, negatively affects the display update rate of the hardware.

  For example, if you set **frameRateInterval()** to 1.0, then the live display frame rate is based on the number of frames acquired and displayed over the last second.

useSoftwareLiveDisplay

```
bool useSoftwareLiveDisplay () const ;

void useSoftwareLiveDisplay (bool on) ;
```

- `bool useSoftwareLiveDisplay () const ;`

  Returns the state of the *useSoftwareLiveDisplay* flag. This flag is false for all frame grabbers.

- `void useSoftwareLiveDisplay (bool on) ;`

  If true, disables the hardware implementation of live display.

  Parameters

  on True enables software live display and disables hardware live display.

Notes

This method is supported only on the MVS-8200 with CVM3, the only platform to support hardware live display.

displayOutput

```
bool displayOutput() const;
void displayOutput(bool on);
```

- `bool displayOutput() const;`

  Gets the *displayOutput* flag. This flag controls whether all acquired images are displayed or not displayed during live display.

- `void displayOutput(bool on);`

  Sets the *displayOutput* flag.

  Parameters

on	If true, all acquired images are displayed. If false, none of the acquired images are displayed. However, acquisition still occurs.

restartDelay

```
double restartDelay() const ;
void restartDelay(double seconds) ;
```

- `double restartDelay() const ;`

  Returns the amount of delay that is inserted between the end of one acquisition and the start of the next.

  Notes

  The default value of 0.0 means that no delay is inserted, allowing live display to run at the maximum frame rate possible.

- `void restartDelay(double seconds) ;`

  Sets the amount of delay that is inserted between the end of one acquisition and the start of the next.

  Parameters

seconds	The delay in seconds. Must be greater than or equal to 0.0.

Notes

You can use the restart delay to control the acquisition/display frame rate of the internal live display thread. The delay is implemented as a non-blocking sleep function call that does not block other threads in the system.

Use caution when using this function, as increasing the delay also reduces the frame rate.

---

makeLocal

```
bool makeLocal() const ;

void makeLocal(bool on) ;
```

- `bool makeLocal() const ;`

Returns the *makeLocal* flag. True means that the image is stored locally. False (the default) means that the image is stored remotely. *Local storage* refers to the memory space of the processor. On a frame grabber, memory is local in the host system and remote on the frame grabber.

In the case of an MVS-8120, it is generally desirable to use local images (in PC memory) for processing and remote images (on the frame grabber) for live display. Live images can be displayed at a higher rate if they are DMAed, or sent directly from the frame grabber to the video card without having to go through PC host memory.

Notes

The *makeLocal* property is currently supported only on the MVS-8120 frame grabber.

If **makeLocal()** returns false, then any user-supplied color map is not used when displaying images.

- `void makeLocal(bool on) ;`

Sets the *makeLocal* flag.

Parameters

on	If true, the pel buffer is stored locally (in PC memory). If false, the pel buffer is stored remotely (on the frame grabber).

Notes

Internally, this method sets the *makeLocal* parameter to **fifo->complete()**. This parameter controls whether the pel buffer returned by this method is local or remote.

The *makeLocal* property is currently supported only on the MVS-8120 frame grabber. Setting this property to false greatly improves live display performance on the MVS-8120.

The *makeLocal* property is supported only when the magnification (zoom) of the display is set to 1.0. This is true when **ccDisplay::mag()** and **ccDisplay::magExact()** both return 1.0. When the magnification is other than 1.0, this parameter is ignored and the internal display system determines the best type of pel buffer, local or remote, for the application.

See the reference page for the **ccPelBuffer** class for more information on the proper use of remote or local pel buffers.

See *When Do Color Maps Apply* and *Live Display Using One FIFO on an MVS-8120* in the *Displaying Images* chapter of the *CVL User's Guide* for more information on the *makeLocal* property.

threadPriority

```
cePriority threadPriority() const;

void threadPriority(cePriority priority);
```

Note These methods are not supported on any MVS-8200 platform.

- `cePriority threadPriority() const;`

  Retrieves the priority of the live display thread.

- `void threadPriority(cePriority priority);`

  Sets the priority of the live display thread.

  Parameters

  priority The thread priority.

  Notes

  The priority of the live display thread is set on the first call to **ccDisplay::startLiveDisplay()**. Subsequent calls to this method do not change the thread priority, unless all live displays on all display objects are stopped.

  If **ccDisplay::stopLiveDisplay()** is called on all display objects, then the next call to **ccDisplay::startLiveDisplay()** sets the new thread priority.

pelbufferCallback

```
const ccCallback1Ptrh& pelbufferCallback() const;

void pelbufferCallback(const ccCallback1Ptrh& callback);
```

- `const ccCallback1Ptrh& pelbufferCallback() const;`

  Retrieves the pel buffer callback. If set, this pel buffer callback is called every time a pel buffer is acquired during live display.

  Notes

  The callback is passed a void pointer that points to the acquired pel buffer.

  See *Using a Live Display Callback* in the Displaying Images chapter of the *CVL User's Guide* for sample code demonstrating how to use a pel buffer callback with live display.

- `void pelbufferCallback(const ccCallback1Ptrh& callback);`

  Sets the pel buffer callback to be called each time an image is acquired during live display.

  Parameters

  callback The callback object.

  Notes

  The supplied callback must take a void pointer that points to the acquired pel buffer.

  The callback, consisting of an overridden **operator()**, should be as fast as possible to avoid adversely affecting the live display frame rate. Cognex does not recommend placing image processing tasks that take a significant amount of time to run inside the callback, as this processing would occur at the same priority as the live display thread.

ccLock

```
#include <ch_cvl/threads.h>

class ccLock;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A class sets a lock on a mutex, semaphore, or event. You set a lock by constructing a **ccLock** and supplying the resource to lock.

In most cases, you should declare your **ccLock** objects as automatic local variables; there is generally no reason to instantiate global or static **ccLock** objects.

Because the **ccLock** destructor is guaranteed to unlock the resource, this class is useful for ensuring that resources are unlocked even if an exception is thrown.

Constructors/Destructors

ccLock

```
ccLock(cc_Resource& m, bool locked=true);

~ccLock();
```

- `ccLock(cc_Resource& m, bool locked=true);`

Construct a **ccLock** object to access the given mutex, semaphore, or event. You can construct a lock without locking the resource by specifying false for the *locked* argument to the constructor.

Parameters

m	The resource. *m* must be derived from **cc_Resource**.
locked	If true, *m* is locked immediately. If false, *m* is not locked. If *locked* is true (the default), the current thread will block until the specified resource is available.

- `~ccLock();`

  Unlocks the resource supplied at construction. If this **ccLock** has been locked more times than it has been unlocked, the missing unlock operations are performed and the resource is unlocked.

Public Member Functions

lock `bool lock(double timeout=HUGE_VAL);`

Locks the resource supplied to this **ccLock** at construction. This function returns true if the resource becomes available, false if the timeout expires.

Parameters

timeout The number of seconds to block. If you specify *HUGE_VAL* for *timeout*, the thread will wait forever.

Throws

cc_Resource::BrokenLock
The **cc_Resource::breakLocks()** static function was invoked on this thread.

lockOrElse `void lockOrElse(double timeout);`

This function is identical to **lock()** except that it throws *cc_Resource::Timeout* if the timeout is exceeded.

Parameters

timeout The number of seconds to wait for the lock. If *timeout* is HUGE_VAL, then the lock waits forever. *timeout* must be greater than or equal to 0.

Throws

cc_Resource::BrokenLock
The **cc_Resource::breakLocks()** static function was invoked on this thread.

cc_Resource::Timeout
The timeout was exceeded.

unlock `void unlock();`

Unlocks the resource supplied to this **ccLock** at construction.

ccLSLineFitter

```
#include <ch_cvl/cfit.h>

class ccLSLineFitter;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class computes the line that produces the best least-squares fit for a set of points.

Notes

This class is deprecated and is maintained for backward compatibility only. It is replaced by the new global function **cfLineFit()**.

Constructors/Destructors

ccLSLineFitter

```
ccLSLineFitter ();

ccLSLineFitter (const cmStd vector<cc2Vect> & points);
```

- `ccLSLineFitter ();`

  Constructs a **ccLSLineFitter** that does not contain any points.

- `ccLSLineFitter (const cmStd vector<cc2Vect> & points);`

  Constructs a **ccLSLineFitter** and initializes it with the supplied points.

  Parameters

  points The points to which to fit a line.

Operators

operator*

```
friend ccLSLineFitter operator* (
  const cc2Xform & pointsXform, const ccLSLineFitter & S);
```

If you multiply a **ccLSLineFitter** by a **cc2Xform**, all of the points in the resulting **ccLSLineFitter** will have been transformed by the supplied **cc2Xform**, and the line produced by this **ccLSLineFitter** is similarly updated.

Parameters

pointsXform The **cc2Xform** by which to multiply this **ccLSLineFitter**

S The **ccLSLineFitter**.

operator+=

```
ccLSLineFitter& operator+=(const ccLSLineFitter& rhs);
```

Adds the points in *rhs* to the current object. This operation is equivalent to, but much more efficient than, calling **update()** repeatedly for each added point.

Parameters

rhs A **ccLSLineFitter** object containing the point set you wish to add to this object.

operator+

```
ccLSLineFitter operator+(const ccLSLineFitter& rhs) const;
```

Returns a **ccLSLineFitter** whose point set includes the *rhs* point set and the point set in this object.

Parameters

rhs The **ccLSLineFitter** object containing the point set you wish to add.

Public Member Functions

reset

```
void reset ();
```

Removes all the points from this **ccLSLineFitter**.

update

```
void update (const cc2Vect & point);
```

Updates this **ccLSLineFitter** to account for a new point. The next call to **fit()** returns a line that reflects the added point.

Parameters

point The point to add.

numPoints

```
int numPoints () const;
```

Returns the number of points accounted for by this **ccLSLineFitter**.

fit

```
ccFLine fit () const;
bool fit(ccFLine& line) const;
```

- `ccFLine fit () const;`

  Returns the computed line that is the best least squares fit of the point set contained in this object.

  The fitter requires at least 2 distinct points

  Throws

  ccLSLineFitter::UnstableErr
  There are fewer than 2 points in this **ccLSLineFitter** or the solution is singular.

- `bool fit(ccFLine& line) const;`

  This is equivalent to the function above except it returns the computed line in *line*, and never throws.

  The function returns false if the point set count is < 2, or if the fit is singular. Otherwise, returns true.

  Parameters

  line The object where the computed line is placed.

error

```
double error (const ccFLine & line) const;
```

Returns the summed absolute error between the points in this **ccLSLineFitter** and the supplied line.

Parameters

line The line to use for computing the error.

fitAndError

```
bool fitAndError(ccFLine* pLine, double* pError) const;
```

Computes the best fit line and the associated error. The computed line is placed in *pLine*, and the computed error is placed in *pError*.

Returns false if the stored point set count is < 2, or if the fit is singular. Otherwise, returns true.

pLine and/or *pError* may be 0 if not needed. (**Note**: with *pLine* = 0, this function provides a faster way to compute the error).

Parameters

pLine	Where the tool places the computed line.
pError	Where the tool places the error. (see **error()**).

ccLSPointToLineFitter

```
#include <ch_cvl/cfit.h>

class ccLSPointToLineFitter;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class computes the transformation that provides the best fit between one or more points and one or more corresponding lines, as determined by the least-squares error.

Constructors/Destructors

ccLSPointToLineFitter

```
ccLSPointToLineFitter ();

ccLSPointToLineFitter (
  const cmStd vector<cc2Vect> & points,
  const cmStd vector<ccFLine> & lines);

ccLSPointToLineFitter (const ccRect & rect,
  const cmStd vector<cc2Vect> &topPoints,
  const cmStd vector<cc2Vect> &bottomPoints,
  const cmStd vector<cc2Vect> &leftPoints,
  const cmStd vector<cc2Vect> &rightPoints,
  bool isLeftHanded = false);
```

- `ccLSPointToLineFitter ();`

  Constructs a **ccLSPointToLineFitter** which contains no points.

- ```
  ccLSPointToLineFitter (
    const cmStd vector<cc2Vect> & points,
    const cmStd vector<ccFLine> & lines);
  ```

  Constructs a **ccLSPointToLineFitter** with the supplied sets of points and lines. Each point in *points* is assumed to correspond to the corresponding line in *lines*.

  Each of the lines in *lines* must be parallel to the x-axis or the y-axis.
  ```

Parameters

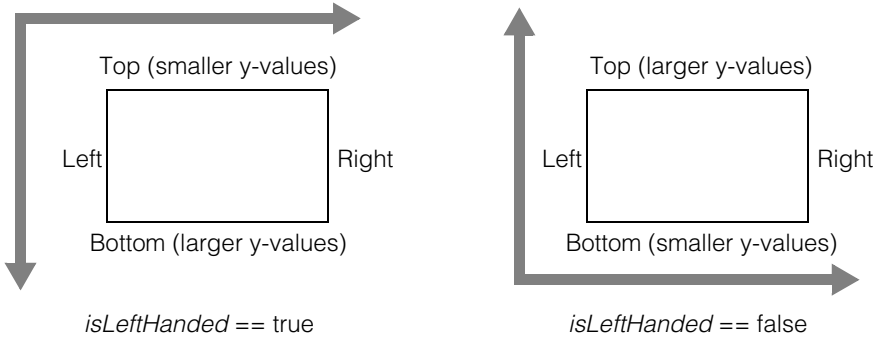
points	The points.
lines	The lines

Throws

ccLSPointToLineFitter::BadArgErr
The size of *lines* does not equal the size of *points,* or one or more of the lines in *lines* is parallel to neither the x-axis nor the y-axis.

-
```
ccLSPointToLineFitter (const ccRect & rect,
  const cmStd vector<cc2Vect> &topPoints,
  const cmStd vector<cc2Vect> &bottomPoints,
  const cmStd vector<cc2Vect> &leftPoints,
  const cmStd vector<cc2Vect> &rightPoints,
  bool isLeftHanded = false);
```

Constructs a **ccLSPointToLineFitter** with the supplied sets of points and lines. Each of the four sets of points is assumed to correspond to a line coincident with one of the sides of the supplied rectangle. The handedness of the points' coordinate system is determined by the *isLeftHanded* argument, as shown below:



Parameters

rect	The rectangle to fit to the supplied points.
topPoints	The points to fit to the top of *rect*.
bottomPoints	The points to fit to the bottom of *rect*.
leftPoints	The points to fit to the left side of *rect*.
rightPoints	The points to fit to the right side of *rect*.

isLeftHanded If true, the top of *rect* is in the negative y-direction from the center and the bottom of *rect* is in the positive y-direction. If false, the top of *rect* is in the positive y-direction from the center and the bottom of *rect* is in the negative y-direction.

Operators

operator*

```
friend ccLSPointToLineFitter operator* (
  const cc2Xform & pointsXform,
  const ccLSPointToLineFitter & S);
```

If you multiply a **ccLSPointToLineFitter** by a **cc2Xform**, all of the points in the resulting **ccLSPointToLineFitter** will have been transformed by the supplied **cc2Xform**. Using this operator is much faster than transforming the points individually.

Parameters

pointsXform The **cc2Xform** by which to multiply this **ccLSPointToLineFitter**

S The **ccLSPointToLineFitter**.

Public Member Functions

reset

```
void reset ();
```

Removes all of the points and lines from this **ccLSPointToLineFitter**.

---

update

```
void update (const cmStd vector<cc2Vect> & points,
  const ccFLine & line);

void update (const cc2Vect & point, ccFLine & line);

void update (const ccRect & rect,
  const cmStd vector<cc2Vect> & topPoints,
  const cmStd vector<cc2Vect> & bottomPoints,
  const cmStd vector<cc2Vect> & leftPoints,
  const cmStd vector<cc2Vect> & rightPoints,
  bool isLeftHanded = false);
```

-

```
void update (const cmStd vector<cc2Vect> & points,
  const ccFLine & line);
```

Updates this **ccLSPointToLineFitter** to include an additional line and corresponding set of points. The next call to **fit()** returns a transformation that reflects the added points and line.

Parameters

points	The points to add.
line	The line to add.

- `void update (const cc2Vect & point, ccFLine & line);`

  Updates this **ccLSPointToLineFitter** to include an additional point-line pair. The next call to **fit()** returns a transformation that reflects the added point and line.

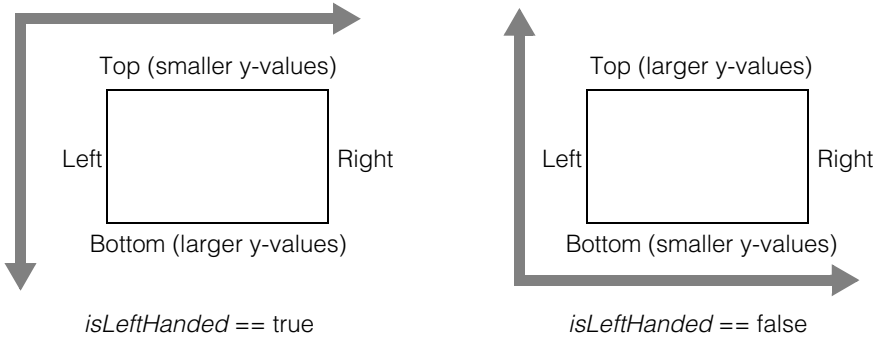
  Parameters

point	The point to add.
line	The line to add.

- ```
  void update (const ccRect & rect,
    const cmStd vector<cc2Vect> & topPoints,
    const cmStd vector<cc2Vect> & bottomPoints,
    const cmStd vector<cc2Vect> & leftPoints,
    const cmStd vector<cc2Vect> & rightPoints,
    bool isLeftHanded = false);
  ```

  Updates this **ccLSPointToLineFitter** to include an additional four lines (specified as being coincident with the sides of a **ccRect**) and four additional sets of points. The next call to **fit()** returns a transformation that reflects the added lines and points.

  Each of the four sets of points is assumed to correspond to a line coincident with one of the sides of the supplied rectangle. The handedness of the points' coordinate system is determined by the *isLeftHanded* argument, as shown below:



  Parameters

rect	The rectangle to fit to the supplied points.
  ```

topPoints	The points to fit to the top of *rect*.
bottomPoints	The points to fit to the bottom of *rect*.
leftPoints	The points to fit to the left side of *rect*.
rightPoints	The points to fit to the right side of *rect*.
isLeftHanded	If true, the top of *rect* is in the negative y-direction from the center and the bottom of *rect* is in the positive y-direction. If false, the top of *rect* is in the positive y-direction from the center and the bottom of *rect* is in the negative y-direction.

numPoints `int numPoints () const;`

Returns the number of point-line correspondences in this **ccLSPointToLineFitter**.

numXPoints `int numXPoints () const;`

Returns the number of point-line correspondences in this **ccLSPointToLineFitter** where the line is parallel to the x-axis.

numYPoints `int numYPoints () const;`

Returns the number of point-line correspondences in this **ccLSPointToLineFitter** where the line is parallel to the y-axis.

fit `cc2Xform fit (int dof);`

Returns the linear transformation that best fits the points to the corresponding lines, given the specified degrees of freedom.

Parameters

dof	The degrees of freedom to enable. The value of *dof* determines which degrees of freedom are enabled as follows: if *dof* is 3, x- and y-translation and rotation are enabled; if *dof* is 4, x- and y-translation, rotation, and uniform scale are enabled; if *dof* is 6, x- and y-translation, rotation, uniform scale, aspect ratio, and skew are enabled.

Notes

The transformation is computed taking into account all point-to-line correspondences that have been added to this **ccLSPointToLineFitter**.

You must have specified a minimum number of point-line correspondences for this **ccLSPointToLineFitter** before calling **fit()**. The minimum number of points depends on the value you specify for *dof*.

- If you specify 3 for *dof*, you must supply at least three point-line correspondences, at least one of them must have a line parallel to the x-axis, and at least one of them must have a line parallel to the y-axis.
- If you specify 4 for *dof*, you must supply at least two point-line correspondences that have a line parallel to the x-axis and at least two point-line correspondences that have a line parallel to the y-axis.
- If you specify 6 for *dof*, you must supply at least three point-line correspondences that have a line parallel to the x-axis and at least three point-line correspondences that have a line parallel to the y-axis.

Throws

ccLSPointToLineFitter::UnstableErr
The solution is either unstable or singular, or an insufficient number of point-line correspondences were supplied.

ccLSPointToLineFitter::BadArgErr
dof is not 3, 4, or 6.

error

```
double error (const cc2Xform & linesTpoints) const;
```

Returns the summed squared error of all the point-line correspondences defined for this **ccLSPointToLineFitter** after applying the supplied transformation to the first set of points.

Parameters

linesTpoints The transformation for which to compute the error.

errorX

```
double errorX (const cc2Xform & linesTpoints) const;
```

Returns the summed squared error of all the point-line correspondences with a line parallel to the x-axis defined for this **ccLSPointToLineFitter** after applying the supplied transformation to the first set of points.

Parameters

linesTpoints The transformation for which to compute the error.

errorY

```
double errorY (const cc2Xform & linesTpoints) const;
```

Returns the summed squared error of all the point-line correspondences with a line parallel to the y-axis defined for this **ccLSPointToLineFitter** after applying the supplied transformation to the first set of points.

Parameters

linesTpoints The transformation for which to compute the error.

fitRect

```
static double fitRect (int dof, const ccRect & rect,
  const cmStd vector<cc2Vect> & topPoints,
  const cmStd vector<cc2Vect> & bottomPoints,
  const cmStd vector<cc2Vect> & leftPoints,
  const cmStd vector<cc2Vect> & rightPoints,
  cc2Xform & pointsTrect, bool isLeftHanded = false);
```

Computes the transformation that best fits the supplied rectangle to the four supplied sets of points, given the supplied degrees of freedom.

Parameters

dof The degrees of freedom to enable. The value of *dof* determines which degrees of freedom are enabled as follows:

If *dof* is 3, x- and y-translation and rotation are enabled.

If *dof* is 4, x- and y-translation, rotation, and uniform scale are enabled.

If *dof* is 6, x- and y-translation, rotation, uniform scale, aspect ratio, and skew are enabled.

rect The rectangle to fit to the supplied points.

topPoints The points to fit to the top of *rect*.

bottomPoints The points to fit to the bottom of *rect*.

leftPoints The points to fit to the left side of *rect*.

rightPoints The points to fit to the right side of *rect*.

pointsTrect A reference to a **cc2Xform** in which the computed transformation is placed.

isLeftHanded If true, the top of *rect* is in the negative y-direction from the center and the bottom of *rect* is in the positive y-direction. If false, the top of *rect* is in the positive y-direction from the center and the bottom of *rect* is in the negative y-direction.

Notes

This is a static member function. Calling it on an instantiated **ccLSPointToLineFitter** has no effect on that **ccLSPointToLineFitter**.

ccLSPointToPointFitter

```
#include <ch_cvl/cfit.h>

class ccLSPointToPointFitter;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class computes the transformation that provides the best fit between two sets of points, as determined by the least-squares error.

Constructors/Destructors

ccLSPointToPointFitter

```
ccLSPointToPointFitter ();

ccLSPointToPointFitter (
  const cmStd vector<cc2Vect> & set1Points,
  const cmStd vector<cc2Vect> & set2Points);
```

- ```
  ccLSPointToPointFitter ();
  ```

  Constructs a **ccLSPointToPointFitter** which contains no points.

- ```
  ccLSPointToPointFitter (
    const cmStd vector<cc2Vect> & set1Points,
    const cmStd vector<cc2Vect> & set2Points);
  ```

  Constructs a **ccLSPointToPointFitter** with the supplied sets of points. Each point in *set1Points* is assumed to correspond to the corresponding point in *set2points*.

  Parameters

  set1Points The first set of points.

  set2Points The second set of points.

  Throws

  ccLSPointToPointFitter::BadArgErr
  The size of *set1Points* is not the same as the size of *set2Points*.

Operators

operator*

```
friend ccLSPointToPointFitter operator* (
  const cc2Xform & set1Xform,
  const ccLSPointToPointFitter & S);
```

If you multiply a **ccLSPointToPointFitter** by a **cc2Xform**, all of the points in the first set of points in the resulting **ccLSPointToPointFitter** will have been transformed by the supplied **cc2Xform**. Using this operator is much faster than transforming the points individually.

Parameters

set1Xform	The **cc2Xform** by which to multiply this **ccLSPointToPointFitter**
S	The **ccLSPointToPointFitter**.

Public Member Functions

reset

```
void reset ();
```

Removes all of the points from this **ccLSPointToPointFitter**.

update

```
void update (const cc2Vect & set1Point,
  const cc2Vect & set2Point);
```

Updates this **ccLSPointToPointFitter** to include an additional pair of points. The next call to **fit()** returns a transformation that reflects the added points.

Parameters

set1Point	The point to add to the first set of points.
set2Point	The point to add to the second set of points.

numPoints

```
int numPoints () const;
```

Returns the number of pairs of points in this **ccLSPointToPointFitter**.

fit

```
cc2Xform fit (int dof);
```

Returns the linear transformation that best fits the points in the second set of points to the points in the first set of points, given the specified degrees of freedom.

Parameters

dof	The degrees of freedom to enable. The value of *dof* determines which degrees of freedom are enabled as follows:

If *dof* is 3, x- and y-translation and rotation are enabled.

If *dof* is 4, x- and y-translation, rotation, and uniform scale are enabled.

If *dof* is 5, x- and y-translation, rotation, uniform scale, and aspect ratio are enabled.

If *dof* is 6, x- and y-translation, rotation, uniform scale, aspect ratio, and skew are enabled.

Notes

The transformation is computed taking into account all point-to-point correspondences that have been added to this **ccLSPointToPointFitter**.

You must have specified a minimum number of points for this **ccLSPointToPointFitter** before calling **fit()**. The minimum number of points depends on the value you specify for *dof*.

- If you specify 3 or 4 for *dof*, you must supply at least 2 points.
- If you specify 5 or 6 for *dof*, you must supply at least 3 points.

Throws

ccLSPointToPointFitter::UnstableErr
The solution is either unstable or singular.

ccLSPointToPointFitter::BadArgErr
dof is not 3, 4, 5, or 6.

error

```
double error (const cc2Xform & set1Tset2) const;
```

Returns the summed squared error between all the point-point correspondences defined for this **ccLSPointToPointFitter** after applying the supplied transformation to the first set of points. Using this function is much faster than applying the transformation to each point in the set, then computing the error.

Parameters

set1Tset2 The transformation for which to compute the error.

ccLSPointToPointFitter

ccMemoryArchive

```
#include <ch_cvl/archive.h>

class ccMemoryArchive : public ccArchive;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	Not applicable

The **ccMemoryArchive** class is a concrete class derived from **ccArchive** that implements a memory archive.

Constructors/Destructors

ccMemoryArchive

```
ccMemoryArchive(
  ccArchive::Ordering ord = ccArchive::eLittleEndian,
  bool seekable=true);

ccMemoryArchive(const char* buf, long length,
  bool seekable=true, bool own=false);

~ccMemoryArchive();
```

- ```
  ccMemoryArchive(
    ccArchive::Ordering ord =ccArchive::eLittleEndian,
    bool seekable=true);
  ```

  Constructs a **ccMemoryArchive** in storing mode. The archive will allocate memory as objects are stored to it.

  Parameters

ord	The endianness of this archive. *ord* must be one of the following values: *ccArchive::eBigEndian* *ccArchive::eLittleEndian*
seekable	If true, the seek position within this archive can be specified.
  ```

- `ccMemoryArchive(const char* buf, c_Int32 length, bool seekable=true, bool own=false);`

Constructs a **ccMemoryArchive** in loading mode using the storage that you supply. If *own* is nonzero the storage buffer will be deleted by this **ccMemoryArchive** when the **ccMemoryArchive** is destroyed.

Parameters

buf	The memory which contains the archive. *buf* must not be null.
length	The number of bytes in the archive. *length* must be greater than or equal to 0.
seekable	If true, the seek position within this archive can be specified.
own	Specifies whether the **ccMemoryArchive** owns the storage pointed to by *buf*. If *own* is nonzero, the storage buffer will be deleted by this **ccMemoryArchive** when the **ccMemoryArchive** is destroyed.

Throws

ccArchive::UnknownArchiveVersion
The loading archive was stored with a later version.

Notes

You must not modify the memory pointed to by *buf* nor delete *buf* while this archive exists.

Public Member Functions

storage

```
const char* storage() const;
```

Returns a pointer to this **ccMemoryArchive**'s current storage. You can call the **ccArchive::tell()** function to find out this **ccMemoryArchive**'s current position; for a storing archive, this is the number of bytes that have been stored into the archive.

Notes

The pointer returned by this function is only valid until the next storing operation on this **ccMemoryArchive**.

You should not delete the storage pointed to by the pointer returned by this function.

ccMouseBite

```
#include <ch_cvl/pmifproc.h>

class ccMouseBite : public cc_FeatureRange;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that describes a single mousebite. A mousebite is a set of boundary points within a matched feature that have the following attributes: the boundary points are contiguous within the feature, and the distance between the boundary points in the run-time image and the corresponding points in the trained image exceeds the standard deviation of all boundary points from their ideal locations in the feature by a specified factor.

You do not instantiate **ccMouseBite**s yourself. You can obtain a vector of **ccMouseBite**s by calling **cfDetectMouseBites()**.

To obtain information about the location of a mousebite, use the functions in the **cc_FeatureRange** class, from which **ccMouseBite** is derived.

Constructors/Destructors

ccMouseBite

```
ccMouseBite();
```

Constructs a **ccMouseBite**.

ccMouseBite

ccMouseEvent

```
#include <ch_cvl/uievent.h>

class ccMouseEvent;
```

Class Properties

Copyable	No
Derivable	Not intended
Archiveable	Complex

This class encapsulates information associated with a mouse event. Mouse events include mouse button clicking and mouse movement.

See the *Displaying Graphics* chapter of the *CVL User's Guide* for a discussion of mouse events associated with using **ccDisplayConsole** windows.

Constructors/Destructors

ccMouseEvent

```
ccMouseEvent();
```

Enumerations

Event

This enumeration specifies the possible reasons for a mouse event.

Value	Meaning
eNone	No key.
eMovement	The mouse moved.
eDownClick	The left mouse button was pressed down.
eDoubleClick	The left mouse button was double clicked.
eUpClick	The left mouse button was released.
eDwell	Currently dwelling.

Button This enumeration specifies the mouse button associated with this mouse event.

Value	Meaning
eNoButton = 0	There is no button associated with this event.
eLeft = 1	The left mouse button.
eMid = 2	The middle mouse button.
eRight = 4	The right mouse button.

Key This enumeration specifies a keyboard key associated with this mouse event.

Value	Meaning
eNoKey = 0	There is no key associated with this mouse event.
eAlt = 1	The **Alt** key.
eControl = 2	The **Ctrl** key.
eShift = 4	The **Shift** key.

Public Member Functions

event

```
void event(Event e);
Event event() const;
```

Mouse events must be one of the **Event** enums.

- `void event(Event e);`

  Sets a new mouse event type.

  Parameters

  e The new mouse event.

- `Event event() const;`

  Returns the mouse event.

whichButton

```
void whichButton(Button b);
Button whichButton() const;
```

Specifies a mouse button associated with this mouse event. Must be one of the **Button** enums.

- `void whichButton(Button b);`

  Sets a button type.

  Parameters

  b The new button type.

- `Button whichButton() const;`

  Returns the button type.

buttons

```
void buttons(c_UInt32 b);
c_UInt32 buttons() const;
```

Specifies the button(s) associated with this mouse event. The mouse buttons are represented by individual bits so that more than one button can be specified in the same event.

Left mouse button = 1
Middle mouse button = 2
Right mouse button = 4

- `void buttons(c_UInt32 b)`

  Sets a new buttons specifier.

  Parameters

  b The new buttons specifier.

- `c_UInt32 buttons() const;`

  Returns the button specifier.

keys

```
void keys(c_UInt32 k);
c_UInt32 keys() const;
```

Specifies the keyboard key(s) associated with this mouse event. The keyboard keys are represented by individual bits so that more than one key can be specified in the same event.

Alt key = 1
Ctrl key = 2
Shift key = 4

- `void keys(c_UInt32 k);`

  Sets a new key word.

  Parameters

k	The new key word.

- `c_UInt32 keys() const;`

  Returns the key word.

position

```
void position(const ccIPair& p);
const ccIPair& position() const;
```

Specifies the mouse position at the time of the event.

- `void position(const ccIPair& p);`

  Sets a new mouse position.

  Parameters

p	The new mouse position.

- `const ccIPair& position() const;`

  Returns the current mouse position setting.

ccMovePartCallbackProp

```
#include <ch_cvl/prop.h>

class ccMovePartCallbackProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

As of CVL 6.1, this class is one of two ways to register a callback class for the move-part state:

Method	Features	See
ccAcqFifo:: movePartInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	*movePartInfoCallback* on page 230
ccMovePartCallbackProp	Calls your callback function.	this section

If you are writing a new CVL application, consider using **ccAcqFifo::movePartInfoCallback()** to register your move-part callback function.

This class describes the move-part callback property of an acquisition FIFO queue. An acquisition enters the movable state (**ccAcqFifo::isMovable()** returns true) when the camera's field of view can be changed without affecting the acquired image. At this point, the acquisition software invokes the move-part callback function that you register with this property.

The callback function you write should set flags or semaphores in your application that allow your program to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition software, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

In general, an acquisition enters the movable state during the acquisition's next-to-last vertical blank interval. With an RS-170 camera, for example, an acquisition would enter the movable state 17 ms (one field time) before it enters the complete state.

Not all frame grabbers can detect the next-to-last vertical blank interval. When using one of these frame grabbers, the acquisition enters the movable and complete states at the same time. In such cases, the move-part callback is always invoked first, followed immediately by the completion callback.

On the MVS-8100M, MVS-8100C, and MVS-8100C/CPCI (but not the MVS-8100M+ when using CCF-based video formats), you can select between default and optimum timing of the move-part callback function's execution. See **movePartTimingChoice** on page 210.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

Constructors/Destructors

ccMovePartCallbackProp

```
ccMovePartCallbackProp();

explicit ccMovePartCallbackProp(ccCallbackPtrh& callback);
```

- `ccMovePartCallbackProp();`

  Creates a new move-part callback property not associated with any FIFO. The default callback function is an unbound handle, meaning no callback function is invoked.

- `explicit ccMovePartCallbackProp(ccCallbackPtrh& callback);`

  Creates a new move-part callback property not associated with any FIFO, registering the function pointed to by *callback* as the move-part callback function.

  Parameters

  callback — Effectively, the callback function to be invoked when the acquisition process enters the move-part state. This is generally when the image has been captured by the camera, but before the image has finished being transferred into local memory. An unbound handle means no callback will occur. Technically, *callback* is a pointer handle to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

Public Member Functions

movePartCallback

```
void movePartCallback(const ccCallbackPtrh& callback);
const ccCallbackPtrh& movePartCallback() const;
```

- `void movePartCallback(ccCallbackPtrh& callback);`

  callback points to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

  Parameters

callback	Effectively, the callback function to be invoked when the acquisition process enters the move-part state. This is generally when the image has been captured by the camera, but before the image has finished being transferred into local memory. An unbound handle means no callback will occur. Technically, *callback* is a pointer handle to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

- `const ccCallbackPtrh& movePartCallback() const;`

  Returns the callback function to be invoked when the acquisition enters the move-part state.

ccMovePartCallbackProp

ccMutex

```
#include <ch_cvl/threads.h>

class ccMutex: public cc_Resource;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A resource that can only be locked by a single thread at a time. Any number of other threads can attempt to lock a mutex, but only one thread can have it locked at a time. A single thread can lock a mutex more than once.

Constructors/Destructors

ccMutex

```
ccMutex(bool locked = false);
```

Construct a new **ccMutex**. You can specify whether this **ccMutex** is locked or unlocked at construction.

Parameters

locked — If true, this **ccMutex** is locked. If false, it is unlocked.

Public Member Functions

lock

```
bool lock(double timeout=HUGE_VAL);
```

Locks this **ccMutex**. Any other thread that attempts to lock this **ccMutex** will block until this **ccMutex** is released. This function returns true if it was able to lock this **ccMutex**. If this **ccMutex** is already locked and remains so for the duration of the specified timeout, this function returns false.

Parameters

timeout — The number of seconds to block. If you specify *HUGE_VAL* for *timeout*, the thread will wait forever or until the **ccMutex** becomes available.

Throws

cc_Resource::BrokenLock — The **cc_Resource::breakLocks()** static function was invoked on this thread.

unlock `void unlock();`

Unlocks this **ccMutex**.

ccOCAlphabet

```
#include <ch_cvl/oc.h>

class ccOCAlphabet : public ccPersistent, public ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class contains the set of character models from which you can construct a line of characters. It also incorporates a confusion score table, which indicates the probability that character pairs may be confused with each other during verification. An optical verification tool uses the information provided by this class to perform optical verification of the characters.

Constructors/Destructors

ccOCAlphabet

```
ccOCAlphabet();

ccOCAlphabet(
  const cmStd vector<ccOCModelPtrh_const>& characters,
  double confusionThreshold = 0.5,
  bool compile = true,
  const ccCvlString& name = cmT(""));

ccOCAlphabet(
  const cmStd vector<ccOCModelPtrh>& characters,
  double confusionThreshold = 0.5,
  bool compile = true,
  const ccCvlString& name = cmT(""));

~ccOCAlphabet();

ccOCAlphabet(const ccOCAlphabet& a);
```

- `ccOCAlphabet();`

  Constructs a nameless, compiled alphabet with no characters. The confusion threshold defaults to 0.5.

- ```
  ccOCAlphabet(
    const cmStd vector<ccOCModelPtrh_const>& characters,
    double confusionThreshold = 0.5,
    bool compile = true,
    const ccCvlString& name = cmT(""));
  ```

  Constructs a named alphabet from the supplied character models.

  Parameters

  characters The character models from which to create the alphabet.

  confusionThreshold
  The confusion threshold for the characters in the alphabet. The confusion threshold determines whether characters are easily confused with each other during optical verification. If the confusion score for a character pairing is greater than the confusion threshold, the two characters are easily confused. The confusion threshold has a range of 0 to 1.

  compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

  name The name of the alphabet.

  Throws

  ccOCDefs::BadParams
  confusionThreshold is less than 0 or greater than 1.

  ccOCDefs::SameKey
  Two or more characters have the same key. Each key in the alphabet must be unique.

- ```
  ccOCAlphabet(
    const cmStd vector<ccOCModelPtrh>& characters,
    double confusionThreshold = 0.5,
    bool compile = true,
    const ccCvlString& name = cmT(""));
  ```

  Constructs a named alphabet from the supplied character models.

  Parameters

  characters The character models from which to create the alphabet.

confusionThreshold
The confusion threshold for the characters in the alphabet. The confusion threshold determines whether characters are easily confused with each other during optical verification. If the confusion score for a character pairing is greater than the confusion threshold, the two characters are easily confused. The confusion threshold has a range of 0 to 1.

compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

name The name of the alphabet.

Throws

ccOCDefs::BadParams
confusionThreshold is less than 0 or greater than 1.

ccOCDefs::SameKey
Two or more characters have the same key. Each key in the alphabet must be unique.

- `~ccOCAlphabet();`

  Destroys this alphabet.

- `ccOCAlphabet(const ccOCAlphabet& a);`

  Creates a new alphabet by copying the specified alphabet.

  Parameters
  a The alphabet to copy.

Operators

operator= `ccOCAlphabet& operator=(const ccOCAlphabet& a);`

Assignment operator.

Parameters
a The source alphabet for the assignment.

operator==

```
bool operator==(const ccOCAlphabet& other) const;
```

Returns true if this alphabet is equal to the specified alphabet.

Parameters

other The alphabet with which to compare for equality.

operator!=

```
bool operator!=(const ccOCAlphabet& other) const;
```

Returns true if this alphabet is not equal to the specified alphabet.

Parameters

other The alphabet with which to compare for inequality.

Public Member Functions

name

```
const ccCvlString& name() const;

void name(const ccCvlString& n);
```

- ```
  const ccCvlString& name() const;
  ```

  Returns the name of this alphabet.

- ```
  void name(const ccCvlString& n);
  ```

  Sets the name of this alphabet.

  Parameters

  n A name string.

characters

```
const cmStd vector<ccOCModelPtrh_const>&
  characters() const;

void characters(
  const cmStd vector<ccOCModelPtrh_const>& chars,
  bool compile = true);

void characters(const cmStd vector<ccOCModelPtrh>& chars,
  bool compile = true);
```

- ```
  const cmStd vector<ccOCModelPtrh_const>&
    characters() const;
  ```

  Returns the characters in this alphabet.
  ```

- ```
  void characters(
    const cmStd vector<ccOCModelPtrh_const>& chars,
    bool compile = true);
  ```

  Sets the characters in this alphabet.

  Parameters

  chars The characters that the alphabet comprises.

  compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

  Throws

  ccOCDefs::SameKey
  Two or more characters have the same key. Each key in the alphabet must be unique.

- ```
  void characters(const cmStd vector<ccOCModelPtrh>& chars,
    bool compile = true);
  ```

  Sets the characters in this alphabet.

  Parameters

  chars The characters that the alphabet comprises.

  compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

  Throws

  ccOCDefs::SameKey
  Two or more characters have the same key. Each key in the alphabet must be unique.

confusionThreshold

```
double confusionThreshold() const;
void confusionThreshold(double t);
```

- `double confusionThreshold() const;`

  Returns the confusion threshold.

- `void confusionThreshold(double t);`

  Sets the confusion threshold for the characters in the alphabet. If the computed confusion score for a character pairing is greater than the confusion threshold, the characters are easily confused with each other during optical verification.

  Parameters

  t The confusion threshold. The valid range is between 0 and 1.

  Notes

  Changing the confusion threshold does not necessitate recompiling the confusion scores for pairings of characters in the alphabet.

  Throws

  ccOCDefs::BadParams

  confusionThreshold is less than 0 or greater than 1.

numChars

```
c_Int32 numChars() const;
```

Returns the number of character models in this alphabet.

hasCharacter

```
bool hasCharacter(c_Int32 key) const;
```

Returns true if a character with the specified key exists in this alphabet.

Parameters

key The key of the character to find.

character

```
const ccOCModelPtrh_const& character(c_Int32 key) const;
```

Returns the character with the specified key.

Parameters

key The key of the character to find.

Throws

ccOCDefs::BadKey
There is no character with the supplied key.

add

```
void add(const ccOCModelPtrh_const& c,
 bool compile = true);

void add(const cmStd vector<ccOCModelPtrh_const>& chars,
 bool compile = true);

void add(const cmStd vector<ccOCModelPtrh>& chars,
 bool compile = true);
```

- ```
  void add(const ccOCModelPtrh_const& c,
   bool compile = true);
  ```

Adds the specified character to the alphabet.

Parameters

c The character to add to the alphabet.

compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

Throws

ccOCDefs::SameKey
The supplied character has the same key as a character that already exists in the alphabet. Each key in the alphabet must be unique. If the function throws an error, the alphabet remains unchanged.

- ```
  void add(const cmStd vector<ccOCModelPtrh_const>& chars,
   bool compile = true);
  ```

Adds the specified characters to the alphabet.

Parameters

chars The characters to add to the alphabet.

compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

Throws

ccOCDefs::SameKey

Two or more of the supplied characters have the same key or the key already exists in the alphabet. Each key in the alphabet must be unique. If the function throws an error, the alphabet remains unchanged.

- ```
  void add(const cmStd vector<ccOCModelPtrh>& chars,
    bool compile = true);
  ```

Adds the specified characters to the alphabet.

Parameters

chars The characters to add to the alphabet.

compile Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

Throws

ccOCDefs::SameKey

Two or more of the supplied characters have the same key or the key already exists in the alphabet. Each key in the alphabet must be unique. If the function throws an error, the alphabet remains unchanged.

remove

```
void remove(c_Int32 key);

void remove(const cmStd vector<c_Int32>& keys);
```

- ```
  void remove(c_Int32 key);
  ```

Removes the character with the specified key from the alphabet. If the alphabet was previously compiled, it remains compiled.

Parameters

key The key of the character to remove.

Throws

ccOCDefs::BadKey

The character specified by the key does not exist in this alphabet.

- `void remove(const cmStd vector<c_Int32>& keys);`

  Removes the characters with the specified keys from the alphabet. If the alphabet was previously compiled, it remains compiled.

  Parameters

  keys The keys of characters to remove.

  Throws

  ccOCDefs::BadKey
  The characters with the specified keys do not exist in this alphabet.

  ccOCDefs::SameKey
  Two or more supplied keys are the same.

confusion `double confusion(c_Int32 key1, c_Int32 key2) const;`

Returns the confusion score for the pairing of characters given by the keys. The confusion score, a number between 0 and 1, indicates the probability that the characters can be confused with one another during optical verification. A returned value of 0 means that the characters will never be confused with each other; a value of 1 means that they will always be confused.

Parameters

key1 The first key, which selects a character from a row of the confusion table.

key2 The second key, which selects a character from a column of the confusion table.

Notes

Confusion scores are based on comparing the characters at the same position, angle, and scale.

Throws

ccOCDefs::BadKey
The specified character keys do not exist in this alphabet.

ccOCDefs::NotCompiled
Confusion scores have not been compiled for characters.

areConfusable

```
bool areConfusable(c_Int32 key1, c_Int32 key2) const;
```

Returns true if the two characters referred to by *key1* and *key2* are easily confused. Two characters are easily confused if their confusion score is greater than the confusion threshold.

Parameters

key1 The first key, which selects a character from a row of the confusion table.

key2 The second key, which selects a character from a column of the confusion table.

Throws

ccOCDefs::BadKey
The specified character keys do not exist in this alphabet.

ccOCDefs::NotCompiled
Confusion scores have not been compiled for characters.

compile

```
void compile(ccDiagObject* diagObject = 0,
  c_UInt32 diagFlags = 0);
```

Computes the confusion score for all possible pairings of characters in the alphabet.

Parameters

diagObject A valid diagnostic object or null.

diagFlags Any diagnostic flags.

Notes

Calling **compile()** has no effect (other than re-generating diagnostic information if you have supplied a value for *diagObject*) if confusion scores are already compiled.

isCompiled

```
bool isCompiled() const;
```

Returns true if the alphabet has compiled confusion scores for all character pairs.

Typedefs

ccOCAlphabetPtrh

```
typedef ccPtrHandle<ccOCAlphabet> ccOCAlphabetPtrh;
```

ccOCAlphabetPtrh_const

```
typedef ccPtrHandle_const<ccOCAlphabet>
  ccOCAlphabetPtrh_const;
```

Deprecated Members

confusionThreshold

```
void confusionThreshold(double t, bool compile);
```

This function is deprecated. You should not use it in new applications.

Sets the confusion threshold for the characters in the alphabet. If the computed confusion score for a character pairing is greater than the confusion threshold, the characters are easily confused with each other during optical verification.

Parameters

t — The confusion threshold. The valid range is between 0 and 1.

compile — Determines whether the alphabet should compile confusion scores for each pairing of characters. Each confusion score indicates the probability that one character can be confused with another. If true, the alphabet compiles the confusion scores for all character pairings at construction-time.

Notes

Changing the confusion threshold does not necessitate recompiling the confusion scores for pairings of characters in the alphabet.

Throws

ccOCDefs::BadParams

confusionThreshold is less than 0 or greater than 1.

ccOCAlphabet

ccOCChar

```
#include <ch_cvl/occhar.h>

class ccOCChar;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCChar** class contains data associated with characters in OCR/OCV. This includes the character codes, images, and other metadata.

Constructors/Destructors

ccOCChar

```
ccOCChar();

ccOCChar(const ccOCChar& rhs);
```

- `ccOCChar();`

  Constructs a character with a default constructed key, a polarity of *eUnknown*, and metrics equal to **ccCharMetrics::blankMetrics()**.

- `ccOCChar(const ccOCChar& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

  Notes

  Compiler-generated destructor is used.

  Copy operator performs deep copy (for example, deep copy of the contained images).

Public Member Functions

reset

```
void reset();
```

Resets the character to a default state.

key

```
const ccOCCharKey& key() const;
void key(const ccOCCharKey& k);
```

- `const ccOCCharKey& key() const;`

  Gets a character key.

- `void key(const ccOCCharKey& k);`

  Sets a character key.

  Parameters

k	The character key.

characterCode

```
c_Char32 characterCode() const;
void characterCode(c_Char32 c);
```

- `c_Char32 characterCode() const;`

  Gets the character code of the character's key.

- `void characterCode(c_Char32 c);`

  Sets the character code of the character's key.

  Parameters

c	The character code.

  Notes

  The setter sets the value within the key.

name32

```
void name32(const ccCvlString32& n);
const ccCvlString32& name32() const;
```

- `void name32(const ccCvlString32& n);`

  Sets the name for this character in Unicode UTF-32.

  Parameters

  n The name to be set.

- `const ccCvlString32& name32() const;`

  Gets the name for this character in Unicode UTF-32.

  Notes

  name32() and **name()** are always kept in sync; therefore, setting one also changes the other one appropriately.

name

```
void name(const ccCvlString& n);
const ccCvlString& name() const;
```

- `void name(const ccCvlString& n);`

  Sets the name for this character.

  Parameters

  n The name to be set.

- `const ccCvlString& name() const;`

  Gets the name for this character.

  Throws

  ccCharCodeDefs::NotRepresentable
  Any character in the underlying UTF-32 representation of the name cannot be represented as a TCHAR.

  Notes

  name32() and **name()** are always kept in sync; therefore, setting one also changes the other one appropriately.

description32

```
void description32(const ccCvlString32& n);
const ccCvlString32& description32() const;
```

- `void description32(const ccCvlString32& n);`

  Sets the description for this character in Unicode UTF-32.

  Parameters

  n The description to be set.

- `const ccCvlString32& description32() const;`

  Gets the description for this character in Unicode UTF-32.

  Notes

  description32() and **description()** are always kept in sync; therefore, setting one also changes the other one appropriately.

description

```
void description(const ccCvlString& n);
const ccCvlString& description() const;
```

- `void description(const ccCvlString& n);`

  Sets the description for this character.

  Parameters

  n The description to be set.

- `const ccCvlString& description() const;`

  Gets the description for this character.

  Throws

  ccCharCodeDefs::NotRepresentable
  Any character in the underlying UTF-32 representation of the description cannot be represented as a TCHAR.

  Notes

  description32() and **description()** are always kept in sync; therefore, setting one also changes the other one appropriately.

polarity

```
ccOCDefs::Polarity polarity() const;

void polarity(ccOCDefs::Polarity p);
```

- `ccOCDefs::Polarity polarity() const;`

  Gets the polarity for this character.

- `void polarity(ccOCDefs::Polarity p);`

  Sets the polarity for this character.

  Parameters

  p The polarity to be set.

  Throws

  ccOCDefs::BadParams
  The polarity is not valid.

image

```
void image(const ccPelBuffer_const<c_UInt8>& image,
 const ccPelRect& maxRootCopyRect = ccPelRect());

ccPelBuffer_const<c_UInt8> image(
 bool rewindowToImageArea = false) const;
```

imageMarkRect

```
ccPelBuffer_const<c_UInt8> imageMarkRect() const;
```

- `void image(const ccPelBuffer_const<c_UInt8>& image, const ccPelRect& maxRootCopyRect = ccPelRect());`

  Sets the image for this character.

  Parameters

  image The image.

  Throws

  ccOCDefs::BadParams
  maxRootCopyRect is not a null rectangle and does not contain the image's window.

Notes

It makes a deep copy of the image, including pixels from the root image outside the image window, if so specified.

maxRootCopyRect may be a null rectangle to indicate that only the image's window should be copied; otherwise, it should enclose the image's window. To copy the entire root, use **image.windowEntire()** for *maxRootCopyRect*.

- ```
  ccPelBuffer_const<c_UInt8> image(
    bool rewindowToImageArea = false) const;
  ```

Gets the image for this character.

Throws

ccOCDefs::BadParams

rewindowToImageArea is true and
a) the image is bound and
b1) the image's window does not contain **imageArea()**, or
b2) **imageArea()** is null.

Notes

Returns a shallow copy of the image.

Returns an unbound image buffer if no image has been specified for this character.

If *rewindowToImageArea* is true, the returned image's window will be set to that returned by **imageArea()**.

- ```
  ccPelBuffer_const<c_UInt8> imageMarkRect() const;
  ```

Gets the image for this character.

Throws

ccOCDefs::RectNotSpecified

The mark rectangle is unspecified in the metrics.

Notes

Returns a shallow copy of the image.

Returns an unbound image buffer if no image has been specified for this character.

mask

```
void mask(const ccPelBuffer_const<c_UInt8>& mask,
 const ccPelRect& maxRootCopyRect = ccPelRect());

ccPelBuffer_const<c_UInt8> mask(
 bool rewindowToImageArea = false) const;
```

maskMarkRect

```
ccPelBuffer_const<c_UInt8> maskMarkRect() const;
```

- ```
  void mask(const ccPelBuffer_const<c_UInt8>& mask,
   const ccPelRect& maxRootCopyRect = ccPelRect());
  ```

  Sets the mask for this character.

  Parameters

  mask The mask.

  Throws

  ccOCDefs::BadParams

  maxRootCopyRect is not a null rectangle and does not contain the image's window.

  Notes

  It makes a deep copy of the image, including pixels from the root image outside the image window, if so specified.

  maxRootCopyRect may be a null rectangle to indicate that only the image's window should be copied; otherwise, it should enclose the image's window. To copy the entire root, use **mask.windowEntire()** for *maxRootCopyRect*.

- ```
  ccPelBuffer_const<c_UInt8> mask(
   bool rewindowToImageArea = false) const;
  ```

  Gets the mask for this character.

  Throws

  ccOCDefs::BadParams

  rewindowToImageArea is true and
  a) the mask is bound and
  b1) the mask's window does not contain imageArea(), or
  b2) imageArea() is null.

Notes

It returns a shallow copy of the mask image.

It returns an unbound image buffer if no mask has been specified for this character.

If *rewindowToImageArea* is true and the mask image is bound, the returned image's window will be set to that returned by **imageArea()**.

- `ccPelBuffer_const<c_UInt8> maskMarkRect() const;`

Gets the mask for this character.

Throws

ccOCDefs::RectNotSpecified
The mark rectangle is unspecified in the metrics.

Notes

It returns a shallow copy of the mask image.

It returns an unbound image buffer if no mask has been specified for this character.

normalizedImage

```
void normalizedImage(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPelRect& maxRootCopyRect = ccPelRect());

ccPelBuffer_const<c_UInt8> normalizedImage(
  bool rewindowToImageArea = false) const;
```

normalizedImageMarkRect

```
ccPelBuffer_const<c_UInt8> normalizedImageMarkRect()
  const;
```

- `void normalizedImage(`
  `const ccPelBuffer_const<c_UInt8>& image,`
  `const ccPelRect& maxRootCopyRect = ccPelRect());`

Sets the normalized image for this character.

Parameters

image The normalized image.

Throws

ccOCDefs::BadParams

maxRootCopyRect is not a null rectangle and does not contain the image's window.

Notes

It makes a deep copy of the image, including pixels from the root image outside the image window, if so specified.

maxRootCopyRect may be a null rectangle to indicate that only the image's window should be copied; otherwise, it should enclose the image's window. To copy the entire root, use **image.windowEntire()** for *maxRootCopyRect*.

- ```
  ccPelBuffer_const<c_UInt8> normalizedImage(
    bool rewindowToImageArea = false) const;
  ```

Gets the normalized image for this character.

Throws

ccOCDefs::BadParams

rewindowToImageArea is true and
a) the normalized image is bound and
b1) the normalized image's window does not contain **imageArea()**, or
b2) **imageArea()** is null.

Notes

It returns a shallow copy of the image.

It returns an unbound image buffer if no normalized image has been specified for this character.

If *rewindowToImageArea* is true and the normalized image is bound, the returned image's window will be set to that returned by **imageArea()**.

- ```
  ccPelBuffer_const<c_UInt8> normalizedImageMarkRect()
    const;
  ```

Gets the normalized image for this character.

Throws

ccOCDefs::RectNotSpecified

The mark rectangle is unspecified in the metrics.

Notes

It returns a shallow copy of the image.

It returns an unbound image buffer if no normalized image has been specified for this character.

normalizedImage

```
void normalizedImage(
  const ccPelBuffer_const<c_UInt8>& image,
  bool hasThresholdAndInvert,
  c_Int32 threshold,
  bool invert,
  const ccPelRect& maxRootCopyRect = ccPelRect());
```

Sets the normalized image for this character and also sets the *threshold* and *invert* values to allow producing a binarized image. A deep copy of the image is made, including pixels from the root image outside the image window, if so specified.

Parameters

image The normalize image to be set.

hasThresholdAndInvert
Whether it has threshold and invert values.

threshold The threshold value.

invert The invert value.

Notes

maxRootCopyRect may be a null rectangle to indicate that only the image's window should be copied; otherwise, it should enclose the image's window. To copy the entire root, use **image.windowEntire()** for *maxRootCopyRect*.

Throws

ccOCDefs::BadParams
maxRootCopyRect is not a null rectangle and does not contain the image's window; or
threshold < 0 or *threshold* > 256

hasThresholdAndInvert

```
bool hasThresholdAndInvert() const;
void hasThresholdAndInvert(bool h);
```

- `bool hasThresholdAndInvert() const;`

  Gets whether the character has threshold and invert values specified to allow binarizing the normalized image.

- `void hasThresholdAndInvert(bool h);`

  Sets whether the character has threshold and invert values specified to allow binarizing the normalized image.

  Parameters

  h Whether the character has threshold and invert values.

  Notes

  See documentation for **ccBlobParams::setSegmentationHardThresh()** in <*ch_cvl/blobtool.h*> for the description of *threshold* and *invert*.

threshold

```
c_Int32 threshold() const;
void threshold(c_Int32 t);
```

- `c_Int32 threshold() const;`

  Gets the threshold value to allow binarizing the normalized image.

  Throws

  ccOCDefs::ThresholdAndInvertNotSpecified
  hasThresholdAndInvert() is false.

- `void threshold(c_Int32 t);`

  Sets the threshold value to allow binarizing the normalized image.

  Parameters

  t The threshold value.

Notes

Setting *threshold* and *invert* values does not also set whether or not the *threshold* and *invert* values are considered specified. You must also explicitly call **hasThresholdAndInvert(***true***)** for these values to be considered specified and thus to allow binarization of the normalized image.

Throws

ccOCDefs::BadParams
t < 0

---

invert

```
bool invert() const;

void invert(bool i);
```

- `bool invert() const;`

Gets the invert value to allow binarizing the normalized image.

Throws

ccOCDefs::ThresholdAndInvertNotSpecified
hasThresholdAndInvert() is false.

- `void invert(bool i);`

Sets the invert value to allow binarizing the normalized image.

Parameters

i The invert value.

Notes

Setting *threshold* and *invert* values does not also set whether or not the *threshold* and *invert* values are considered specified. You must also explicitly call **hasThresholdAndInvert(***true***)** for these values to be considered specified and thus to allow binarization of the normalized image.

---

binarizedImage

```
ccPelBuffer_const<c_UInt8> binarizedImage(
  bool rewindowToImageArea = false) const;
```

binarizedImageMarkRect

```
ccPelBuffer_const<c_UInt8> binarizedImageMarkRect() const;
```

- `ccPelBuffer_const<c_UInt8> binarizedImage(`
  `bool rewindowToImageArea = false) const;`

Gets the binarized image for this character.

Throws

ccOCDefs::ThresholdAndInvertNotSpecified
hasThresholdAndInvert() is false.

ccOCDefs::BadParams
threshold() > 256.

ccOCDefs::BadParams
normalizedImage() returns a bound image, and
hasThresholdAndInvert() is true, and
rewindowToImageArea is true, and
a) The binarized image's window does not contain **imageArea()**
or
b) **imageArea()** is null.

- `ccPelBuffer_const<c_UInt8> binarizedImageMarkRect() const;`

Gets the binarized image for this character.

Notes

The binarized image is always a hard-thresholded version of the normalized image, produced using the current **threshold()** and **invert()** values. If the *threshold* and *invert* values were not specified, it is an error to try to get the binarized image.

Requires that **hasThresholdAndInvert()** is true.

Returns a shallow copy of the binarized image if **normalizedImage()** returns a bound image; otherwise returns an unbound image.

When returning a bound image, if *rewindowToImageArea* is true, the returned image's window will be set to that returned by **imageArea()**.

Throws

ccOCDefs::ThresholdAndInvertNotSpecified
hasThresholdAndInvert() is false.

ccOCDefs::BadParams
threshold() > 256.

ccOCDefs::BadParams
normalizedImage() returns a bound image, and
hasThresholdAndInvert() is true, and
rewindowToImageArea is true, and
a) The binarized image's window does not contain **imageArea()**
or
b) **imageArea()** is null.

ccOCDefs::RectNotSpecified
The mark rectangle is unspecified in the metrics.

imageArea

```
ccPelRect imageArea() const;
void imageArea(const ccPelRect& area);
```

- `ccPelRect imageArea() const;`

  Gets the character area for this character in image coordinates.

- `void imageArea(const ccPelRect& area);`

  Sets the character area for this character in image coordinates.

  The default is **ccPelRect()**

  Parameters
  area The character area.

markOrImageArea

```
ccPelRect markOrImageArea() const;
```

Returns the image-coordinate mark rectangle if it is specified and non-null, or else returns **imageArea()**.

moveOrigin

```
void moveOrigin(const cc2Vect& origin);
```

Moves the origin to the specified location; that is, makes the specified point become (0, 0) in client coordinates.

Parameters
origin The origin.

metrics

```
const ccOCCharMetrics& metrics() const;
void metrics(const ccOCCharMetrics& m);
```

- `const ccOCCharMetrics& metrics() const;`

  Gets the metrics for this character.

- `void metrics(const ccOCCharMetrics& m);`

  Sets the metrics for this character.

  Parameters

m	The metrics to be set.

makeUniqueInstanceKey

```
static ccOCCharKey makeUniqueInstanceKey(
  const ccOCCharKey& key,
  const cmStd vector<ccOCChar>& characters);

static ccOCCharKey makeUniqueInstanceKey(
  const ccOCCharKey& key,
  const cmStd vector<ccOCCharKey>& keys);
```

- ```
  static ccOCCharKey makeUniqueInstanceKey(
    const ccOCCharKey& key,
    const cmStd vector<ccOCChar>& characters);
  ```

  Returns a unique key to be used for a new instance of the given key that does not match any of the other characters provided, with the same character code, variant, and font ID.

  Parameters

key	The given key.
characters	The provided characters.

- ```
  static ccOCCharKey makeUniqueInstanceKey(
    const ccOCCharKey& key,
    const cmStd vector<ccOCCharKey>& keys);
  ```

  Returns a unique key to be used for a new instance of the given key that does not match any of the other keys provided, with the same character code, variant, and font ID.

  Parameters

key	The given key.
keys	The provided keys.

makeUniqueInstanceKeys

```
static bool makeUniqueInstanceKeys(
  cmStd vector<ccOCCharKey>& keys);
```

Makes all keys unique by assigning new instance values if necessary, with the same character code, variant, and font ID.

Parameters

keys The keys.

Returns true if any keys had their instance values changed or false otherwise.

Operators

operator=

```
ccOCChar& operator=(const ccOCChar& rhs);
```

Assignment operator.

Parameters

rhs The object to assign to this one.

Notes

Assignment operator performs deep copy.

operator==

```
bool operator==(const ccOCChar& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCChar& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCCharKey

```
#include <ch_cvl/occhar.h>

class ccOCCharKey;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a character key, specifying the full identity of a character. The key consists of a Unicode UTF-32 code point as well as an instance number, a variant number, and a font ID.

Constructors/Destructors

ccOCCharKey

```
explicit ccOCCharKey(
  c_Char32 characterCode = ccCharCode::eUnknown,
  c_Int32 instance = eUnspecified,
  c_Int32 variant = eUnspecified,
  c_Int32 fontID = eUnspecified);
```

Constructor.

Parameters

characterCode	The character code.
instance	The instance.
variant	The variant.
fontID	The font ID.

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Enumerations

enum

```
enum { eUnspecified = -1 };
```

Special value for a key to indicate that a field is unspecified. This value may be used for the instance, variant, and/or font ID of a key. Note that this is NOT the same value as *ccCharCode::eUnknown*, which is the corresponding value for a character code.

Public Member Functions

reset

```
void reset();
```

Resets the key to a default state.

See the default constructor for default values.

isCharacterCodeKnown

```
bool isCharacterCodeKnown() const;
```

Returns whether the character code is not *ccCharCode::eUnknown*.

characterCode

```
c_Char32 characterCode() const;

void characterCode(c_Char32 c);
```

- `c_Char32 characterCode() const;`

  Gets the character code, which is a Unicode UTF-32 code point.

- `void characterCode(c_Char32 c);`

  Sets the character code, which is a Unicode UTF-32 code point.

  Parameters

c	The character code.

  Notes

  The value can be *ccCharCode::eUnknown* to indicate that the character code is not known.

  It is not recommended to set a character code outside the UTF-32 code range. See <*ch_cvl/charcode.h*>.

isInstanceSpecified

```
bool isInstanceSpecified() const;
```

Returns whether instance is not equal to *eUnspecified*.

instance

```
c_Int32 instance() const;
```

```
void instance(c_Int32 i);
```

- `c_Int32 instance() const;`

  Gets the instance number, which can be used to uniquely identify a particular instance of a character with a given character code within a particular font.

- `void instance(c_Int32 i);`

  Sets the instance number, which can be used to uniquely identify a particular instance of a character with a given character code within a particular font.

  Parameters

  i The instance number.

  For example, a set of characters (including an image for each character) might have two different instances of a character code (for example, the letter “A”) to represent two different appearances that both corresponded to that character code. If that font also had a single instance of the letter “B”, then the character codes and instances might reasonably be the following:

  (characterCode=”A”, instance=0)

  (characterCode=”A”, instance=1)

  (characterCode=”B”, instance=0)

  Notes

  The value can be *eUnspecified* to indicate that the instance is not specified.

isVariantSpecified

```
bool isVariantSpecified() const;
```

Returns whether variant is not equal to *eUnspecified*.

variant

```
c_Int32 variant() const;
void variant(c_Int32 v);
```

- `c_Int32 variant() const;`

  Gets the variant number, which can be used to indicate a distinct appearance of a character code.

- `void variant(c_Int32 v);`

  Sets the variant number, which can be used to indicate a distinct appearance of a character code.

  Parameters

  v The variant number.

  For example, a particular font might contain two visually distinct appearances for the digit zero “0”, one of which might simply be an ellipse and another that is an ellipse with a diagonal stroke through it.

Notes

instance is independent of *variant*; for example, in a given set of characters one would typically have no duplicate *instance* values regardless of whether the characters had different *variant* values.

variant is provided for your convenience, and its interpretation is application-specific. OCR and OCV functions preserve this value but otherwise make no use of it.

The concept of variant is related to the concept of *variant selectors* in the Unicode Standard, but the value of the *variant* does not need to be a Unicode variant selector value.

The value can be *eUnspecified* to indicate that the variant is not specified.

isFontIDSpecified

```
bool isFontIDSpecified() const;
```

Returns whether the font ID is not equal to *eUnspecified*.

fontID

```
c_Int32 fontID() const;
void fontID(c_Int32 id);
```

- `c_Int32 fontID() const;`

  Gets the font ID.

- `void fontID(c_Int32 id);`

  Sets the font ID.

  Parameters

  id The font ID.

Notes

This is the ID of the font (that is, set of characters) of which this character is a member. One potential use for this value would be to have each font use a distinct value for the characters in that font, so that an application could refer to a set of character keys from all the fonts and still be able to distinguish which character came from which font.

fontID is provided for your convenience, and its interpretation is application-specific. OCR and OCV functions preserve this value but otherwise make no use of it.

For applications that choose to use font IDs to keep track of different fonts, the *instance* values should typically be required to be distinct only within a single font and not across all fonts.

The value can be *eUnspecified* to indicate that the font ID is not specified.

isRepresentableAsChar

```
bool isRepresentableAsChar() const ;
```

Returns whether the character code is representable as type char. See <*ch_cvl/charcode.h*>.

isRepresentableAsWChar

```
bool isRepresentableAsWChar() const;
```

Returns whether the character code is representable as type wchar_t. See <*ch_cvl/charcode.h*>.

isRepresentableAsTChar

```
bool isRepresentableAsTChar() const;
```

Returns whether the character code is representable as type TCHAR. See <*ch_cvl/charcode.h*>.

isSpace

```
bool isSpace() const;
```

Returns whether the character code is a space character.

Notes

This test is performed by **ccCharCode::isSpace()**.

This is entirely different and separate from blank metrics (**ccOCCharMetrics::isBlank()**).

equalCharacterCode

```
bool equalCharacterCode(const ccOCCharKey& rhs) const;
bool equalCharacterCode(c_Char32 characterCode) const;
```

- `bool equalCharacterCode(const ccOCCharKey& rhs) const;`

  Returns whether the two keys have the same character code.

  Parameters

  rhs The character key to compare to this one.

- `bool equalCharacterCode(c_Char32 characterCode) const;`

  Returns whether this key has the given character code.

  Parameters

  characterCode The character code.

isMatch

```
bool isMatch(const ccOCCharKey& otherKey) const;
```

Returns whether this key matches the other key.

Parameters

otherKey The other key.

Notes

Two keys match each other if all their corresponding data members match. A match is determined as follows:

For character code:

- The value *ccCharCode::eUnknown* never matches anything, including itself. In other words, two keys do not match if they both have the character code *ccCharCode::eUnknown*.
- Otherwise, two character codes match each other only if they are equal.

For fontID, instance, and variant:

- The value *eUnspecified* matches everything, including itself. For example, an unspecified instance number matches another unspecified instance number; it also matches any other specified instance number.
- Otherwise, two numbers match each other only if they are equal.

Operators

operator char()

```
operator char() const;
```

Convert **characterCode()** to a character.

Throws

ccCharCodeDefs::NotRepresentable
characterCode is not representable as the given type.

operator wchar_t()

```
operator wchar_t() const;
```

Convert **characterCode()** to a wide character.

Throws

ccCharCodeDefs::NotRepresentable
characterCode is not representable as the given type.

operator<

```
bool operator<(const ccOCCharKey& rhs) const;
```

Compares this key to *rhs*. Keys are ordered first by *characterCode*, then by *fontID*, then by instance, then by variant.

Parameters

rhs The base of comparison.

Notes

Character codes that are unknown and other values that are unspecified are considered greater than known/specified values.

operator==

```
bool operator==(const ccOCCharKey& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharKey& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCCharKey

ccOCCharMetrics

```
#include <ch_cvl/occhar.h>

class ccOCCharMetrics;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCCharMetrics contains metadata associated with characters for OCR/OCV.

Constructors/Destructors

ccOCCharMetrics

```
ccOCCharMetrics();
```

Constructor.

As default, all metrics are unspecified. All metrics values are default constructed or zero.

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Enumerations

RectType

```
enum RectType;
```

This enumeration specifies the particular rectangle of interest; that is, cell, mark, and detected mark rectangles, and whether in client or image coordinates.

Value	Meaning
eCellRectClient = 0x01	Cell rectangle in client coordinates. Note: The internal representation is a floating-point rectangle.
eCellRectImage = 0x02	Cell rectangle in image coordinates. Note: The internal representation is a fixed-point rectangle.
eMarkRectClient = 0x04	Mark rectangle in client coordinates. Note: The internal representation is a floating-point rectangle.
eMarkRectImage = 0x08	Mark rectangle in image coordinates. Note: The internal representation is a fixed-point rectangle.
eDetectedMarkRectClient = 0x10	Detected mark rectangle in client coordinates. Note: The internal representation is a floating-point rectangle.
eDetectedMarkRectImage = 0x20	Detected mark rectangle in image coordinates. Note: The internal representation is a fixed-point rectangle.

Public Member Functions

reset

```
void reset();
```

Resets the metrics to a default state.

isRectSpecified

```
bool isRectSpecified(RectType rectType) const;
void isRectSpecified(RectType rectType, bool isSpecified);
```

- `bool isRectSpecified(RectType rectType) const;`

  Gets whether the given type of rectangle is specified.

  Parameters

  rectType The rectangle type.

- `void isRectSpecified(RectType rectType, bool isSpecified);`

  Sets whether the given type of rectangle is specified.

  Parameters

  rectType The rectangle type.

  isSpecified Whether the given type of rectangle is specified.

  Notes

  It is legal for the image-coordinate version of a rectangle (for example, *eCellRectImage*) to be specified but the client-coordinate version of that same rectangle (for example, *eCellRectClient*) not to be specified, or vice versa.

  Throws

  ccOCDefs::BadParams

  rectType is not a valid enum value

rect

```
ccRect rect(RectType rectType) const;
void rect(RectType rectType, const ccRect& r);
void rect(RectType rectType, const ccPelRect& r);
```

- `ccRect rect(RectType rectType) const;`

  Gets the specified type of rectangle as a floating-point rectangle.

  It is legal to get any of the rectangle types using this function, even the image-coordinate rectangles that are stored internally as fixed-point integers.

  Parameters

  rectType The rectangle type.

Throws

ccOCDefs::BadParams
rectType is not a valid enum value.

ccOCDefs::RectNotSpecified
The rectangle of the given type is not specified.

- `void rect(RectType rectType, const ccRect& r);`

Sets the specified type of rectangle as a floating-point rectangle.

Parameters

rectType The rectangle type.

r The rectangle to set.

- `void rect(RectType rectType, const ccPelRect& r);`

Sets the specified type of rectangle as a floating-point rectangle.

Parameters

rectType The rectangle type.

r The rectangle to set.

Notes

It is legal to set any of the rectangle types using this function, even the image-coordinate rectangles that are stored internally as fixed-point integers, as long as the values can be represented without overflow. Note that underflow will not be reported as an error.

Setting a rectangle does not also set whether or not the rectangle is considered specified. The user must also explicitly call **isRectSpecified(***rectType, true***)** for the given rectangle type to be considered specified.

Throws

ccOCDefs::BadParams
rectType is not a valid enum value.

ccOCDefs::IntegerOverflow
Integer overflow occurs.

encloseRect

```
ccPelRect encloseRect(RectType rectType) const;
```

Gets the enclosing rectangle of the specified type of rectangle.

It is legal to get any of the rectangle types using this function, even the client-coordinate rectangles that are stored internally as floating-point numbers.

Parameters

rectType The rectangle type.

Throws

ccOCDefs::BadParams
rectType is not a valid enum value.

ccOCDefs::RectNotSpecified
The rectangle of the given type is not specified.

ccOCDefs::IntegerOverflow
Integer overflow occurs.

encloseRectImage

```
ccPelRect encloseRectImage(RectType rectType,
  const cc2Xform& imageFromClientXform) const;

ccPelRect encloseRectImage(RectType rectType,
  const cc_PelBuffer& image) const;
```

- ```
  ccPelRect encloseRectImage(RectType rectType,
    const cc2Xform& imageFromClientXform) const;
  ```

  Gets the enclosing rectangle of the specified type of rectangle in image coordinates.

  Parameters

  rectType The rectangle type.

  imageFromClientXform
  A **cc2Xform** that specifies the transformation from the image coordinate system.

  Throws

  ccOCDefs::BadParams
  rectType is not a valid enum value.

  ccOCDefs::RectNotSpecified
  The rectangle of the given type is not specified.

  ccOCDefs::IntegerOverflow
  Integer overflow occurs.
  ```

- ```
  ccPelRect encloseRectImage(RectType rectType,
    const cc_PelBuffer& image) const;
  ```

  Gets the enclosing rectangle of the specified type of rectangle in image coordinates.

  For *eCellRectImage*, *eMarkRectImage*, and *eDetectedMarkRectImage*, this function is the same as **encloseRect()**. For *eCellRectClient*, *eMarkRectClient*, and *eDetectedMarkRectClient*, the specified client-coordinate rectangle is first converted to image coordinates using the image's *imageFromClient* xform, and the enclosing rectangle of that rectangle is returned.

  Parameters

  rectType The rectangle type.

  image The image whose *imageFromClient* xform is used for conversion.

  Notes

  It is legal to get any of the rectangle types using this function, even the client-coordinate rectangles that are stored internally as floating-point numbers.

  It is legal for image to be unbound.

  Throws

  ccOCDefs::BadParams
  rectType is not a valid enum value.

  ccOCDefs::RectNotSpecified
  The rectangle of the given type is not specified.

  ccOCDefs::BadParams
  image.imageFromClientXformBase()->**isLinear()** returns false.

  ccOCDefs::IntegerOverflow
  Integer overflow occurs.

setImageRectsFromClientRects

```
void setImageRectsFromClientRects(
  const cc2Xform& imageFromClientXform);

void setImageRectsFromClientRects(
  const cc_PelBuffer& image);
```

- ```
  void setImageRectsFromClientRects(
    const cc2Xform& imageFromClientXform);
  ```

  For each of the client-coordinate rectangle types, set the corresponding image-coordinate rectangle type.

Parameters

imageFromClientXform
A **cc2Xform** that specifies the transformation from the image coordinate system.

Throws

ccOCDefs::BadParams
image.imageFromClientXformBase()->**isLinear()** returns false.

ccOCDefs::IntegerOverflow
Integer overflow occurs.

- ```
  void setImageRectsFromClientRects(
    const cc_PelBuffer& image);
  ```

For each of the client-coordinate rectangle types, set the corresponding image-coordinate rectangle type.

Parameters

image Image that defines the client-coordinates for this rectangle.

Notes

In addition to setting the rectangle data, this also updates the image-coordinate rectangle's **isRectSpecified()** status to be the same as the corresponding client-coordinate rectangle.

It is legal for image to be unbound.

Throws

ccOCDefs::IntegerOverflow
Integer overflow occurs.

---

cellRect

```
ccRect cellRect() const;

void cellRect(const ccRect& r);
```

- ```
  ccRect cellRect() const;
  ```

Gets the client-coordinate cell rectangle.

Throws

ccOCDefs::RectNotSpecified
The rectangle of the given type is not specified.

- `void cellRect(const ccRect& r);`

  Sets the client-coordinate cell rectangle.

  Parameters
  r The client-coordinate cell rectangle.

  See **rect()**.

markRect

```
ccRect markRect() const;
void markRect(const ccRect& r);
```

- `ccRect markRect() const;`

  Gets the client-coordinate mark rectangle.

  Throws
  ccOCDefs::RectNotSpecified
  The rectangle of the given type is not specified.

- `void markRect(const ccRect& r);`

  Sets the client-coordinate mark rectangle.

  Parameters
  r The client-coordinate mark rectangle.

  See **rect()**.

isMarkRectSpecified

```
bool isMarkRectSpecified() const;
void isMarkRectSpecified(bool isSpecified);
```

- `bool isMarkRectSpecified() const;`

  Gets whether the mark rectangle is specified.

- `void isMarkRectSpecified(bool isSpecified);`

  Sets whether the mark rectangle is specified.

  Parameters
  isSpecified Whether the mark rectangle is specified.

Notes

A blank character has a mark rectangle that is specified but degenerate.

hasDetectedMarkRect

```
bool hasDetectedMarkRect() const;
void hasDetectedMarkRect(bool isSpecified);
```

- `bool hasDetectedMarkRect() const;`

  Gets whether there is a detected mark rectangle.

- `void hasDetectedMarkRect(bool isSpecified);`

  Sets whether there is a detected mark rectangle.

  Parameters

  isSpecified Whether there is a detected mark rectangle.

  Notes

  A blank character does not have a detected mark rectangle.

detectedMarkRect

```
ccRect detectedMarkRect() const;
void detectedMarkRect(const ccRect& r);
```

- `ccRect detectedMarkRect() const;`

  Gets the client-coordinate detected mark rectangle.

  Throws

  ccOCDefs::RectNotSpecified
  The rectangle of the given type is not specified.

- `void detectedMarkRect(const ccRect& r);`

  Sets the client-coordinate detected mark rectangle.

  Parameters

  r The client-coordinate detected mark rectangle.

  See **rect()**.

isBlank `bool isBlank() const;`

Gets whether or not the character is a blank.

This is the same as testing for a mark rectangle that is specified and degenerate.

Notes

This is entirely different and separate from the character code being a space character (**ccOCCharKey::isSpace()**).

makeBlank `void makeBlank();`

Sets the character blank.

Making the character blank is the same as specifying a degenerate mark rectangle.

Notes

This is entirely different and separate from the character code being a space character (**ccOCCharKey::isSpace()**).

blankMetrics `static ccOCCharMetrics blankMetrics();`

Returns an object with a specified and degenerate mark rectangle, with all other values default constructed.

isAdvanceSpecified

```
bool isAdvanceSpecified() const;
void isAdvanceSpecified(bool isSpecified);
```

- `bool isAdvanceSpecified() const;`

  Gets whether the advance is specified.

- `void isAdvanceSpecified(bool isSpecified);`

  Sets whether the advance is specified.

  Parameters

  isSpecified Whether the advance is specified.

advance

```
cc2Vect advance() const;
void advance(const cc2Vect& a);
```

- `cc2Vect advance() const;`

  Gets the advance for the character in client coordinates.

  Throws

  ccOCDefs::MetricNotSpecified
  isAdvanceSpecified() returned false.

- `void advance(const cc2Vect& a);`

  Sets the advance for the character in client coordinates.

  Parameters

  a The advance value.

  Notes

  Setting the advance does not also set whether or not the advance is considered specified. You must also explicitly call **isAdvanceSpecified(***true***)** for the advance to be considered specified.

Operators

operator==

```
bool operator==(const ccOCCharMetrics& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

ccOCCharMetrics

ccOCCharSegmentLineResult

```
#include <ch_cvl/ocsegmnt.h>

class ccOCCharSegmentLineResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the segmentation line result, representing an entire line of segmented characters.

Constructors/Destructors

ccOCCharSegmentLineResult

```
ccOCCharSegmentLineResult();

ccOCCharSegmentLineResult(
  const ccOCCharSegmentLineResult& rhs);
```

- `ccOCCharSegmentLineResult();`

  Constructor.

  isFound = false

- `ccOCCharSegmentLineResult(`
  `  const ccOCCharSegmentLineResult& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

  Notes

  Copy performs deep copies of images.

  Compiler-generated destructor is used.

Public Member Functions

reset

```
void reset();
```

Resets this result to a default state.

Notes

This sets *isFound* to false.

isFound

```
bool isFound() const;
```

Returns whether the segmenter found a line of characters.

Notes

All other getters throw *ccOCRDefs::NotComputed* if **isFound()** returns false.

positionResults

```
const cmStd vector<ccOCCharSegmentPositionResult>&
 positionResults() const;
```

Returns results for all segmented characters in this line.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

rectifiedLineImage

```
ccPelBuffer_const<c_UInt8> rectifiedLineImage() const;
```

Returns an image of the line of the segmented characters after rectification.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

Notes

The client coordinates of the *rectifiedLineImage* have uniform scale in x and y. That uniform scale is based on the scale of the original input image. That uniform scale is computed as the square root of the *mapArea* of the original input image.

The client coordinate axes of the *rectifiedLineImage* are approximately parallel to the coordinate axes specified by the search region affine rectangle.

hasNormalizedRectifiedLineImage

```
bool hasNormalizedRectifiedLineImage() const;
```

Returns whether this object contains an image of the line of the segmented characters after rectification and normalization.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

normalizedRectifiedLineImage

```
ccPelBuffer_const<c_UInt8> normalizedRectifiedLineImage()
  const;
```

Returns an image of the line of the segmented characters after rectification and normalization.

Throws

ccOCRDefs::NotComputed
isFound() returns false.

ccOCRDefs::NotComputed
hasNormalizedRectifiedLineImage() returned false.

Notes

The client coordinates of the *normlizedRectifiedLineImage* have uniform scale in x and y. That uniform scale is based on the scale of the original input image. That uniform scale is computed as the square root of the *mapArea* of the original input image.

The client coordinate axes of the *normalizedRectifiedLineImage* are approximately parallel to the coordinate axes specified by the search region affine rectangle.

hasThresholdAndInvert

```
bool hasThresholdAndInvert() const;
```

Gets whether this object has threshold and invert values specified to allow binarizing the normalized image.

Notes

See the documentation for **ccBlobParams::setSegmentationHardThresh()** and the <*ch_cvl/blobtool.h*> header file for the description of *threshold* and *invert*.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

threshold `c_Int32 threshold() const;`

Gets the threshold value used to binarize the normalized image.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

ccOCRDefs::NotComputed
hasThresholdAndInvert() returned false.

invert `bool invert() const;`

Gets the invert value used to binarize the normalized image.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

ccOCRDefs::NotComputed
hasThresholdAndInvert() returned false.

binarizedRectifiedLineImage

```
ccPelBuffer_const<c_UInt8> binarizedRectifiedLineImage()
const;
```

Returns an image of the line of the segmented characters after rectification, normalization, and binzarization.

Throws

ccOCRDefs::NotComputed
isFound() returned false.

ccOCRDefs::NotComputed
hasNormalizedRectifiedLineImage() returned false.

ccOCRDefs::NotComputed
hasThresholdAndInvert() returned false.

Notes

The binarized image is always a hard-thresholded version of the normalized image, produced using the **threshold()** and **invert()** values. If the *threshold* and *invert* values were not specified, it is an error to try to get the binarized image.

The client coordinates of the *binarizedRectifiedLineImage* have uniform scale in x and y. That uniform scale is based on the scale of the original input image. That uniform scale is computed as the square root of the *mapArea* of the original input image.

The client coordinate axes of the *binarizedRectifiedLineImage* are approximately parallel to the coordinate axes specified by the search region affine rectangle.

Operators

operator=

```
ccOCCharSegmentLineResult& operator=(
  const ccOCCharSegmentLineResult& rhs);
```

Assignment operator.

Parameters

rhs The object to assign to this one.

Notes

Assignment performs deep copies of images.

operator==

```
bool operator==(const ccOCCharSegmentLineResult& rhs)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharSegmentLineResult& rhs)
  const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCCharSegmentLineResult

ccOCCharSegmentParagraphResult

```
#include <ch_cvl/ocsegmnt.h>

class ccOCCharSegmentParagraphResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the segmentation paragraph result, representing one or more lines of segmented characters.

Constructors/Destructors

ccOCCharSegmentParagraphResult

```
ccOCCharSegmentParagraphResult();
```

Constructor.

isComputed = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Resets this result to a default state.

Notes

This sets *isComputed* to false.

isComputed

```
bool isComputed() const;
```

Gets whether the result contains computed information.

Notes

All other getters will throw *ccOCRDefs::NotComputed* if **isComputed()** returns false.

lineResults

```
const cmStd vector<ccOCCharSegmentLineResult>&
  lineResults() const;
```

Returns results for all segmented lines in this paragraph.

Throws

ccOCRDefs::NotComputed
isComputed() returned false.

Operators

operator==

```
bool operator==(const ccOCCharSegmentParagraphResult& rhs)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharSegmentParagraphResult& rhs)
  const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCCharSegmentPositionResult

```
#include <ch_cvl/ocsegmnt.h>

class ccOCCharSegmentPositionResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the segmentation position result, representing a single segmented character.

Constructors/Destructors

ccOCCharSegmentPositionResult

```
ccOCCharSegmentPositionResult();
```

Constructor.

isComputed = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Reset this result to a default state.

Notes

This sets *isComputed* to false.

isComputed

```
bool isComputed() const;
```

Gets whether the result contains computed information.

Notes

All other getters throw *ccOCRDefs::NotComputed* if **isComputed()** returns false.

character `const ccOCChar& character() const;`

Gets the segmented character from the result.

Throws

ccOCRDefs::NotComputed
isComputed() returned false.

Notes

The client coordinates of the character have uniform scale in x and y, the same scale as the *rectifiedLineImage*. That uniform scale is based on the scale of the original input image. That uniform scale is computed as the square root of the *mapArea* of the original input image.

isSpace `bool isSpace() const;`

Gets whether the segmented character is a space.

Throws

ccOCRDefs::NotComputed
isComputed() returned false.

spaceScore `double spaceScore() const;`

Gets the score of the segmented character, if the segmented character is a space. The score is in the range [0, 1]. See the documentation for **ccOCCharSegmentSpaceParams::SpaceScoreMode**.

Throws

ccOCRDefs::NotComputed
isComputed() returned false.

ccOCRDefs::NotComputed
isSpace() returned false.

markRect `const ccAffineRectangle& markRect() const;`

Gets the mark rectangle of the segmented character.

The mark rectangle is specified in the client coordinates of the input image used for segmentation. The mark rectangle of a character is a tight bounding box enclosing all of the foreground (for example, ink) pixels in the character image.

Notes

If **isSpace()** is true, the returned affine rectangle is degenerate.

Throws

ccOCRDefs::NotComputed
isComputed() returned false.

cellRect

```
const ccAffineRectangle& cellRect() const;
```

Gets the cell rectangle of the segmented character.

The cell rectangle is specified in the client coordinates of the input image used for segmentation. The cell rectangle is a bounding box that encloses not only all of the foreground (for example, ink) pixels of a character image, but also typically some additional padding region. Cell rectangles are typically the height of the full line of text containing the character. Ideally, all of the cell rectangles in a line of text horizontally touch their adjacent neighbors.

Throws
ccOCRDefs::NotComputed
isComputed() returned false.

Operators

operator==

```
bool operator==(const ccOCCharSegmentPositionResult& rhs)
 const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters
rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharSegmentPositionResult& rhs)
 const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters
rhs The object with which to compare for inequality.

ccOCCharSegmentPositionResult

ccOCCharSegmentResult

```
#include <ch_cvl/ocsegmnt.h>

class ccOCCharSegmentResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the segmentation result, representing one or more paragraphs of segmented characters.

Constructors/Destructors

ccOCCharSegmentResult

```
ccOCCharSegmentResult();
```

Constructor.

isComputed = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Resets this result to a default state.

Notes

This sets *isComputed* to false.

isComputed

```
bool isComputed() const;
```

Gets whether the result contains computed information.

Notes

All other getters throw *ccOCRDefs::NotComputed* if **isComputed()** returns false.

paragraphResults

```
const cmStd vector<ccOCCharSegmentParagraphResult>&
paragraphResults() const;
```

Returns results for all segmented paragraphs.

Throws

ccOCRDefs::NotComputed
isComputed() returns false.

Operators

operator==

```
bool operator==(const ccOCCharSegmentResult& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharSegmentResult& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCCharSegmentRunParams

```
#include <ch_cvl/ocsegmnt.h>

class ccOCCharSegmentRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains runtime parameters for character segmentation.

Character segmentation consists of the following named stages, performed in sequence:

1. Refine line:
   Determine line location, angle, skew, and/or polarity.
2. Normalize:
   Normalize the input image to produce a normalized image. The normalized image should be binarizable using a single global threshold.
3. Binarize:
   Determine a threshold and use it to binarize the normalized image to produce a binarized image.
4. Fragment:
   Perform blob analysis to produce character fragments, where each character fragment is a single blob. See <ch_cvl/blob.h>.
5. Group:
   Group character fragments together to form characters. Grouping may include merging and/or splitting fragments. Small characters may be discarded.
6. Analyze:
   Optionally perform additional analysis to determine a more optimal grouping.

Each runtime parameter is used in one or more stages of segmentation; each parameter notes which stage(s) it is used in with a comment of the form “Stage: Name”, where “Name” is one of the names listed above.

Constructors/Destructors

ccOCCharSegmentRunParams

```
ccOCCharSegmentRunParams();
```

Constructor.

Notes

See individual getters for default values.

Compiler-generated copy constructor, assignment operator, and destructor are used.

Enumerations

NormalizationMode

```
enum NormalizationMode;
```

This enumeration defines how to normalize the image so that the normalized image can be binarized well using only a global threshold.

Value	Meaning
eNormalizationModeNone = 0x1	No normalization is performed; the input image is used as the normalized image.
eNormalizationModeGlobal = 0x2	A global normalization is performed, using only information about the image as a whole, not local variations.
eNormalizationModeLocal = 0x4	A local normalization is performed, using information about each local region of the image to normalize that region.
eNormalizationModeLocalAdvanced = 0x8	A local normalization is performed, using information about each local region of the image to normalize that region, including adjusting not only for the background but also for the contrast of the foreground.
kDefaultNormalizationMode = *eNormalizationModeLocalAdvanced*	The default mode for image normalization.

CharacterFragmentMergeMode

```
enum CharacterFragmentMergeMode;
```

This enumeration defines how to merge character fragments to form characters during the Group stage. Note that the Analyze stage may perform additional merges.

Value	Meaning
eMergeModeRequireOverlap = 0x1	Character fragments must overlap horizontally by at least one pixel to be merged.
eMergeModeSpecifyMinIntercharacterGap = 0x2	Character fragments with a horizontal gap between them may be merged to form characters, where any two fragments with a gap less than the specified minimum intercharacter gap will be merged.
eMergeModeSpecifyGaps = 0x4	Character fragments with a horizontal gap between them may be merged to form characters, with the decision to merge two fragments based on both the specified minimum intercharacter gap and the specified maximum intracharacter gap.
kDefaultMergeMode = *eMergeModeRequireOverlap*	The default is *eMergeModeRequireOverlap*.

AnalysisMode `enum AnalysisMode;`

This enumeration defines the type of analysis to perform to determine the optimal character segmentation.

Value	Meaning
eAnalysisModeMinimal = 0x1	Perform minimal segmentation. This mode performs only straightforward segmentation.
eAnalysisModeStandard = 0x2	Perform standard segmentation, which consists of straightforward segmentation followed by additional analysis to determine the optimal segmentation using the entire line of characters rather than just treating each character individually.
kDefaultAnalysisMode = *eAnalysisModeStandard*	The default mode for character segmentation analysis.

Public Member Functions

reset `void reset();`

Resets all parameters to default values.

Notes

See individual getters for default values.

polarity

```
ccOCDefs::Polarity polarity() const;
void polarity(ccOCDefs::Polarity p);
```

- `ccOCDefs::Polarity polarity() const;`

  Gets the polarity of the characters in the input image.

- `void polarity(ccOCDefs::Polarity p);`

  Sets the polarity of the characters in the input image.

  If the value is *ccOCDefs::eUnknown*, the polarity is automatically determined.

  The default value is *ccOCDefs::eUnknown*.

Parameters

p The polarity.

Throws

ccOCRDefs::BadParams

p is not a valid enum value.

Notes

Stage: Refine line, Normalize

angleHalfRange

```
ccDegree angleHalfRange() const;

void angleHalfRange(const ccDegree& a);
```

- `ccDegree angleHalfRange() const;`

Gets half of the angle search range.

Throws

ccOCRDefs::BadParams

a < 0 degrees or *a* > 180 degrees.

- `void angleHalfRange(const ccDegree& a);`

Sets half of the angle search range.

Parameters

a Half of the angle search range.

Throws

ccOCRDefs::BadParams

a < 0 degrees or *a* > 180 degrees.

Notes

The line angle search range is centered on the *xRotation* of the affine rectangle used as the ROI for the OCR Segmenter tool. In other words, the full search range is [*xRotation* - *angleHalfRange*, *xRotation* + *angleHalfRange*].

Increasing *angleHalfRange* increases the OCR Segmenter tool's run time.

Setting *angleHalfRange* to 0 effectively disables angle search.

Stage: Refine line

The default value is 0 degrees.

skewHalfRange

```
ccDegree skewHalfRange() const;

void skewHalfRange(const ccDegree& s);
```

- `ccDegree skewHalfRange() const;`

  Gets half of the skew search range.

  Throws

  ccOCRDefs::BadParams
  s < 0 degrees or *s* >= 90 degrees.

- `void skewHalfRange(const ccDegree& s);`

  Sets half of the skew search range.

  Parameters

  s Half of the skew range.

  Throws

  ccOCRDefs::BadParams
  s < 0 degrees or *s* >= 90 degrees.

  Notes

  The line skew search range is centered on the skew of the affine rectangle used as the ROI for the OCR Segmenter tool. In other words, the full search range is [*skew* - *skewHalfRange*, *skew* + *skewHalfRange*].

  Increasing *skewHalfRange* increases the OCR Segmenter tool's run time.

  Setting *skewHalfRange* to 0 effectively disables skew search.

  Stage: Refine line

  The default value is 0 degrees.

normalizationMode

```
NormalizationMode normalizationMode() const;

void normalizationMode(NormalizationMode m);
```

- `NormalizationMode normalizationMode() const;`

  Gets the method used to normalize the input image.

- `void normalizationMode(NormalizationMode m);`

  Sets the method used to normalize the input image.

  Parameters

  m The method.

  Throws

  ccOCRDefs::BadParams

  m is not a valid enum value.

  The default value is *ccOCCharSegmentRunParams::kDefaultNormalizationMode*.

  Notes

  Stage: Normalize

useStrokeWidthFilter

```
bool useStrokeWidthFilter() const;

void useStrokeWidthFilter(bool u);
```

- `bool useStrokeWidthFilter() const;`

  Gets whether to remove from the normalized image everything that does not appear to have the same stroke width as the rest of the image.

- `void useStrokeWidthFilter(bool u);`

  Sets whether to remove from the normalized image everything that does not appear to have the same stroke width as the rest of the image.

  This can be useful if, for example, characters appear to be connected to each other by thin noise streaks. However, using the stroke width filter might incorrectly remove real characters if the stroke widths are inconsistent.

  The default value is true.

  Parameters

  u Whether to use stroke width filter.

  Notes

  Stage: Normalize

foregroundThresholdFrac

```
double foregroundThresholdFrac() const;
void foregroundThresholdFrac(double t);
```

- `double foregroundThresholdFrac() const;`

  Gets a modifier in the range [0, 1] that is used to compute the binarization threshold, in the normalized image, that binarizes between foreground and background.

  Throws

  ccOCRDefs::BadParams
  t < 0 or *t* > 1.

- `void foregroundThresholdFrac(double t);`

  Sets a modifier in the range [0, 1] that is used to compute the binarization threshold, in the normalized image, that binarizes between foreground and background.

  Parameters

  t The modifier used to compute the threshold.

  Throws

  ccOCRDefs::BadParams
  t < 0 or *t* > 1.

  - When the input character polarity is *ccOCDefs::eLightOnDark*:

    threshold = round(max((1 - *foregroundThresholdFrac*) * 255, 1))

    invert = false

    where *threshold* and *invert* are used in the same sense as in the Blob tool. (See the documentation for the Blob tool and <*ch_cvl/blobtool.h*>)

    A value of zero results in a binarization threshold value of 255, indicating that only the pixels with the value of 255 (considered most certain to be foreground) should be treated as foreground. A value of one gives a binarization threshold value of 1, indicating that all pixels should be treated as foreground except that those with the value of 0 (considered most certain to be background) should be treated as background.

  - When the input character polarity is *ccOCDefs::eDarkOnLight*:

    threshold = round(max(*foregroundThresholdFrac* * 255, 1))

    invert = true

where *threshold* and *invert* are used in the same sense as in the Blob tool. (See the documentation for the Blob tool and <*ch_cvl/blobtool.h*>)

A value of zero results in a binarization threshold value of 1, indicating that only the pixels with the value of 0 (considered most certain to be foreground) should be treated as foreground. A value of one gives a binarization threshold value of 255, indicating that all pixels should be treated as foreground except that those with the value of 255 (considered most certain to be background) should be treated as background.

In summary, a value of zero indicates that only the pixels considered most certain to be foreground should be treated as foreground, and a value of one indicates that all pixels should be treated as foreground except for those considered most certain to be background.

Notes

As shown by the formulas above, the fraction refers to the greylevel range of the input image type (for example, [0, 255] for an 8-bit greyscale image), and does not refer to an area fraction of the image. In particular, the fraction does not correspond to *threshPercent* in **ccBlobRunParams::setSegmentationHardThresh()** [percent overload]; see *ch_cvl/blobtool.h*.

Stage: Binarize

The default value is 0.5.

characterFragmentMaxDistanceToMainLine

```
double characterFragmentMaxDistanceToMainLine() const;
void characterFragmentMaxDistanceToMainLine(double t);
```

- `double characterFragmentMaxDistanceToMainLine() const;`

  Gets a modifier that is the maximum distance a fragment can have outside the main line of characters as percentage of estimated line height.

- `void characterFragmentMaxDistanceToMainLine(double t);`

  Sets a modifier that is the maximum distance a fragment can have outside the main line of characters as percentage of estimated line height.

  The default value is 0.

  Parameters

t	The modifier to be set.

Notes
Stage: Fragment

ignoreBorderFragments

```
bool ignoreBorderFragments() const;
void ignoreBorderFragments(bool ignore);
```

- `bool ignoreBorderFragments() const;`

  Gets whether to completely ignore any fragments that touch any border of the ROI.

- `void ignoreBorderFragments(bool ignore);`

  Sets whether to completely ignore any fragments that touch any border of the ROI.

  Ignoring such fragments can be useful for excluding non-text features such as the edges of labels that might be included within the ROI.

  The default value is false.

  Parameters

  ignore — Whether to completely ignore any fragments that touch any border of the ROI.

  Notes

  Stage: Fragment

characterFragmentMinNumPels

```
c_Int32 characterFragmentMinNumPels() const;
void characterFragmentMinNumPels(c_Int32 n);
```

- `c_Int32 characterFragmentMinNumPels() const;`

  Gets the minimum number of foreground (that is, text) pixels that a character fragment must have in order to be considered for possible inclusion in a character.

  Throws

  ccOCRDefs::BadParams
  $n < 1$.

- `void characterFragmentMinNumPels(c_Int32 n);`

  Sets the minimum number of foreground (that is, text) pixels that a character fragment must have in order to be considered for possible inclusion in a character.

  A character fragment is a blob in the binarized image. Character fragments that contain fewer foreground pixels than this value are completely ignored for all further processing, as though that fragment had never been detected.

  The default value is 15.

  Parameters

  n The minimum number of foreground.

  Throws

  ccOCRDefs::BadParams
  n < 1.

  Notes

  Stage: Fragment

characterFragmentContrastThreshold

```
double characterFragmentContrastThreshold() const;

void characterFragmentContrastThreshold(double c);
```

- `double characterFragmentContrastThreshold() const;`

  Gets the minimum amount of contrast (in normalized image greylevels) that a fragment must have, relative to the binarization threshold, in order to be considered for possible inclusion in a character.

- `void characterFragmentContrastThreshold(double c);`

  Sets the minimum amount of contrast (in normalized image greylevels) that a fragment must have, relative to the binarization threshold, in order to be considered for possible inclusion in a character.

  Parameters

  c The minimum amount of contrast to be set.

  See the comments for **foregroundThresholdFrac()** for the value of the binarization threshold.

  Any character fragment with a contrast lower than this value is completely ignored for all further processing, as though that fragment had never been detected.

The default value is 30.

Throws

ccOCRDefs::BadParams
c is less than 0.

Notes

Stage: Fragment

characterFragmentMergeMode

```
CharacterFragmentMergeMode characterFragmentMergeMode()
 const;

void characterFragmentMergeMode(
 CharacterFragmentMergeMode m);
```

- `CharacterFragmentMergeMode characterFragmentMergeMode()
 const;`

  Gets the mode used to determine whether to merge two fragments into one character during the Group stage.

- `void characterFragmentMergeMode(
 CharacterFragmentMergeMode m);`

  Sets the mode used to determine whether to merge two fragments into one character during the Group stage.

  Parameters

  m The mode to be set.

Mode values are summarized in the following table.

Value	Meaning
eMergeModeRequireOverlap	Character fragments must overlap horizontally by at least one pixel to be merged. The amount of overlap required is specified by *characterFragmentMinXOverlap*.
eMergeModeSpecifyMinIntercharacterGap	Character fragments with a horizontal gap between them may be merged to form characters, where any two fragments with a gap less than **minIntercharacterGap()** will be merged.
eMergeModeSpecifyGaps	Character fragments with a horizontal gap between them may be merged to form characters, with the decision to merge two fragments based on both **minIntercharacterGap()** and **maxIntracharacterGap()**.

Notes

Using *eMergeModeSpecifyMinIntercharacterGap* is equivalent to using *eMergeModeSpecifyGaps* with *maxIntracharacterGap* = *minIntercharacterGap* - 1.

Stage: Group, Analyze.

The default value is *kDefaultMergeMode.*

characterFragmentMinXOverlap

```
double characterFragmentMinXOverlap() const;

void characterFragmentMinXOverlap(double x);
```

- `double characterFragmentMinXOverlap() const;`

Gets the minimum fraction by which two character fragments must overlap each other in the x-direction for the two fragments to be considered part of the same character.

Throws

ccOCRDefs::BadParams

$x < 0$ or $x > 1$.

- `void characterFragmentMinXOverlap(double x);`

  Sets the minimum fraction by which two character fragments must overlap each other in the x-direction for the two fragments to be considered part of the same character.

  The default value is 0.0.

  Parameters

  x The minimum fraction.

  Throws

  ccOCRDefs::BadParams
  x < 0 or *x* > 1.

  Notes

  This value is used only if *characterFragmentMergeMode* is *eMergeModeRequireOverlap*.

  Stage: Group

characterMinNumPels

```
c_Int32 characterMinNumPels() const;

void characterMinNumPels(c_Int32 n);
```

- `c_Int32 characterMinNumPels() const;`

  Gets the minimum number of foreground (that is, text) pixels that a character must have in order to be reported.

  Throws

  ccOCRDefs::BadParams
  n < 1.

- `void characterMinNumPels(c_Int32 n);`

  Sets the minimum number of foreground (that is, text) pixels that a character must have in order to be reported.

  The default value is 30.

  Parameters

  n The minimum number of foreground pixels.

  Throws

  ccOCRDefs::BadParams
  n < 1.

Notes
Stage: Group, Analyze

characterMinWidth

```
c_Int32 characterMinWidth() const;

void characterMinWidth(c_Int32 w);
```

- `c_Int32 characterMinWidth() const;`

  Gets the minimum width of a character's mark rectangle, in pixels, that a character must have to be reported.

  Throws
  ccOCRDefs::BadParams
  w < 1.

- `void characterMinWidth(c_Int32 w);`

  Sets the minimum width of a character's mark rectangle, in pixels, that a character must have to be reported.

  The default value is 3.

  Parameters
  w The minimum width of a character's mark rectangle.

  Throws
  ccOCRDefs::BadParams
  w < 1.

  Notes
  Stage: Group, Analyze

useCharacterMaxWidth

```
bool useCharacterMaxWidth() const;

void useCharacterMaxWidth(bool u);
```

- `bool useCharacterMaxWidth() const;`

  Gets whether to use *characterMaxWidth* to split wide characters.

- `void useCharacterMaxWidth(bool u);`

  Get/set whether to use *characterMaxWidth* to split wide characters.

  The default value is false.

  Parameters

  u Whether to use *characterMaxWidth*.

  Notes

  Stage: Group, Analyze

characterMaxWidth

```
c_Int32 characterMaxWidth() const;
void characterMaxWidth(c_Int32 m);
```

- `c_Int32 characterMaxWidth() const;`

  Gets the maximum allowable width of a character's mark rectangle in pixels.

  Throws

  ccOCRDefs::BadParams
  m < 1.

- `void characterMaxWidth(c_Int32 m);`

  Sets the maximum allowable width of a character's mark rectangle in pixels.

  A character wider than this value is split into pieces that are not too wide. This value is used only if *useCharacterMaxWidth* is true. The technique used to split the character into pieces depends on the value of *widthType*. If *widthType* is *eFontCharWidthTypeFixed*, the OCR Segmenter tool tries to break the character into pieces of approximately equal width. If *widthType* is *eFontCharWidthTypeVariable* or *eFontCharWidthTypeUnknown*, the OCR Segmenter tool tries to break the character into pieces that are not required to be approximately the same width.

  Parameters

  m The maximum allowable width of a character's mark rectangle.

Notes

The OCR Segmenter tool may sometimes choose to return characters slightly wider than this value if it cannot find a reasonable split.

characterMinAspect is another way of specifying a maximum character width.

Stage: Group, Analyze

The default value is 100.

Throws

ccOCRDefs::BadParams
m < 1.

characterMinHeight

```
c_Int32 characterMinHeight() const;

void characterMinHeight(c_Int32 h);
```

- `c_Int32 characterMinHeight() const;`

  Gets the minimum height of a character's mark rectangle, in pixels, that a character must have to be reported.

  Throws

  ccOCRDefs::BadParams
  h < 1.

- `void characterMinHeight(c_Int32 h);`

  Sets the minimum height of a character's mark rectangle, in pixels, that a character must have to be reported.

  The default value is 3.

  Parameters

  h The minimum height of a character's mark rectangle.

  Throws

  ccOCRDefs::BadParams
  h < 1.

  Notes

  Stage: Group, Analyze

useCharacterMaxHeight

```
bool useCharacterMaxHeight() const;

void useCharacterMaxHeight(bool u);
```

- `bool useCharacterMaxHeight() const;`

  Gets whether to use *characterMaxHeight* to limit the maximum height of a character and/or the line of characters.

- `void useCharacterMaxHeight(bool u);`

  Sets whether to use *characterMaxHeight* to limit the maximum height of a character and/or the line of characters.

  The default value is false.

  Parameters

  u Whether to use *characterMaxHeight*.

  Notes

  Stage: Refine line, Group, Analyze

characterMaxHeight

```
c_Int32 characterMaxHeight() const;

void characterMaxHeight(c_Int32 m);
```

- `c_Int32 characterMaxHeight() const;`

  Gets the maximum allowable height of a character's mark rectangle, in pixels.

  Throws

  ccOCRDefs::BadParams
  g < 1.

- `void characterMaxHeight(c_Int32 m);`

  Sets the maximum allowable height of a character's mark rectangle, in pixels.

  This value is used in two ways. First, this value is used when finding the line as a whole; for example, to reject vertically adjacent noise and/or other lines of vertically adjacent characters. Second, an individual character whose height exceeds this value will be trimmed to meet this height. This value is used only if *useCharacterMaxHeight* is true.

Parameters

m The maximum allowable height of a character's mark rectangle.

Notes

A tall character may be trimmed to a height below *characterMaxHeight* if the OCR Segmenter tool chooses so.

The default value is 100.

Throws

ccOCRDefs::BadParams

g < 1.

Notes

Stage: Refine line, Group, Analyze

minIntercharacterGap

```
c_Int32 minIntercharacterGap() const;

void minIntercharacterGap(c_Int32 g);
```

- `c_Int32 minIntercharacterGap() const;`

  Gets the minimum gap size, in pixels, that can occur between two characters.

  Throws

  ccOCRDefs::BadParams

  g < 0.

- `void minIntercharacterGap(c_Int32 g);`

  Sets the minimum gap size, in pixels, that can occur between two characters.

  If the gap between two fragments is smaller than this, then they must be considered to be part of the same character, unless the combined character would be too wide (as specified by *characterMaxWidth* and/or *characterMinAspect*). The gap is measured from the right edge of the mark rectangle of one character to the left edge of the mark rectangle of the next character.

  This value is used only if *characterFragmentMergeMode* is *eMergeModeSpecifyMinIntercharacterGap* or *eMergeModeSpecifyGaps*.

  The default value is 1.

  Parameters

  g The minimum gap size.

Throws

ccOCRDefs::BadParams
g < 0.

Notes

Stage: Group, Analyze

maxIntracharacterGap

```
c_Int32 maxIntracharacterGap() const;

void maxIntracharacterGap(c_Int32 g);
```

- `c_Int32 maxIntracharacterGap() const;`

  Gets the maximum gap size, in pixels, that can occur within a single character, even for damaged characters.

- `void maxIntracharacterGap(c_Int32 g);`

  Sets the maximum gap size, in pixels, that can occur within a single character, even for damaged characters.

  An intracharacter gap might occur, for example, between successive columns of dots in dot matrix print, or between two pieces of a solid character that was damaged by a scratch. Any gap larger than this value is always interpreted as a break between two separate characters, whereas gaps less than or equal to this value may be interpreted either as a break between two separate characters or as a gap within a single character.

  This value is used only if *characterFragmentMergeMode* is *eMergeModeSpecifyGaps*.

  The default value is 0.

  Parameters

  g Maximum gap size.

  Throws

  ccOCRDefs::BadParams
  g < 0.

  Notes

  Stage: Group, Analyze

useCharacterMinAspect

```
bool useCharacterMinAspect() const;
void useCharacterMinAspect(bool u);
```

- `bool useCharacterMinAspect() const;`

  Gets whether to use *characterMinAspect* to split wide characters.

- `void useCharacterMinAspect(bool u);`

  Sets whether to use *characterMinAspect* to split wide characters.

  The default value is true.

  Parameters

  u Whether to use *characterMinAspect*.

  Notes

  Stage: Group, Analyze

characterMinAspect

```
double characterMinAspect() const;
void characterMinAspect(double m);
```

- `double characterMinAspect() const;`

  Gets the minimum allowable aspect of a character, where the aspect is defined as the height of the entire line of characters divided by the width of the character's mark rectangle.

  Throws

  ccOCRDefs::BadParams
  m < 0.

- `void characterMinAspect(double m);`

  Sets the minimum allowable aspect of a character, where the aspect is defined as the height of the entire line of characters divided by the width of the character's mark rectangle.

  A character whose aspect is smaller than this value (that is, whose width is too large) is split into pieces that are not too wide. This value is used only if *useCharacterMinAspect* is true. The technique used to split the character into pieces depends on the value of *widthType*. If *widthType* is *eFontCharWidthTypeFixed*, the OCR Segmenter tool tries to break the character into pieces of approximately equal width. If *widthType* is *eFontCharWidthTypeVariable* or *eFontCharWidthTypeUnknown*, the OCR Segmenter tool tries to break the character into pieces that are not required to be approximately the same width.

  Parameters

  m The minimum allowable aspect of a character.

  Notes

  The segmenter may sometimes choose to return characters slightly wider than the width indicated by this value if it cannot find a reasonable split.

  characterMaxWidth is another way of specifying a maximum character width.

  This definition of aspect is more convenient than simply using the height of the character's mark rectangle divided by the width of the character's mark rectangle; for example, so that allowing a dash "–" does not require using a tiny *characterMinAspect* value.

  The default value is 0.8.

  Throws

  ccOCRDefs::BadParams
  m < 0.

  Notes

  Stage: Group, Analyze

widthType

```
ccOCRDefs::FontCharWidthType widthType() const;
void widthType(ccOCRDefs::FontCharWidthType w);
```

- `ccOCRDefs::FontCharWidthType widthType() const;`

  Gets how the widths of characters in the font are expected to vary.

  Throws

ccOCRDefs::BadParams
w is not a valid enum value.

- `void widthType(ccOCRDefs::FontCharWidthType w);`

Sets how the widths of characters in the font are expected to vary.

Notes

The character width is the width of the mark rectangle (for example, the bounding box of the ink), not the cell rectangle (which would typically include padding around the mark rectangle).

Stage: Group, Analyze

The default value is *ccOCRDefs::kDefaultFontCharWidthType*.

Parameters

w How the widths of characters in the font are expected to vary.

Throws

ccOCRDefs::BadParams
w is not a valid enum value.

analysisMode

```
AnalysisMode analysisMode() const;
void analysisMode(AnalysisMode a);
```

- `AnalysisMode analysisMode() const;`

Gets whether to perform *minimal analysis* or *standard analysis*.

Throws

ccOCRDefs::BadParams
a is not a valid enum value.

- `void analysisMode(AnalysisMode a);`

Sets whether to perform *minimal analysis* or *standard analysis*.

Minimal analysis performs straightforward segmentation according to the previous parameters. Standard analysis performs an analysis of the line as a whole (including, for example, character spacing) to determine the optimal segmentation.

The default value is *ccOCRDefs::kDefaultAnalysisMode*.

Parameters

a Whether to perform *minimal analysis* or *standard analysis*.

Throws
ccOCRDefs::BadParams
a is not a valid enum value.

Notes
Stage: Analyze

pitchMetric

```
ccOCRDefs::FontPitchMetric pitchMetric() const;
void pitchMetric(ccOCRDefs::FontPitchMetric m);
```

- `ccOCRDefs::FontPitchMetric pitchMetric() const;`

Gets the metric used to specify the spacing of characters.

Throws
ccOCRDefs::BadParams
m is not a valid enum value.

- `void pitchMetric(ccOCRDefs::FontPitchMetric m);`

Sets the metric used to specify the spacing of characters.

Notes
Pitch is the distance between (approximately) corresponding points on adjacent characters and not the distance from the end of one character to the beginning of the next character (which is called the “intercharacter gap”).

Also note that specifying the pitch metric does not necessarily imply that the measured pitch values are expected to be constant.

Stage: Analyze

Parameters
m The metric to specify the spacing of characters.

Notes
This parameter is not used if *analysisMode* is *eAnalysisModeMinimal*.

The default value is *ccOCRDefs::kDefaultFontPitchMetric*.

Throws
ccOCRDefs::BadParams
m is not a valid enum value.

pitchType

```
ccOCRDefs::FontPitchType pitchType() const;
void pitchType(ccOCRDefs::FontPitchType t);
```

- `ccOCRDefs::FontPitchType pitchType() const;`

  Gets how individual pitch values are expected to vary; the pitch values are measured as specified by the pitch metric.

  Throws

  ccOCRDefs::BadParams
  t is not a valid enum value.

- `void pitchType(ccOCRDefs::FontPitchType t);`

  Sets how individual pitch values are expected to vary; the pitch values are measured as specified by the pitch metric.

  Parameters

  t How individual pitch values are expected to vary.

  Notes

  This parameter is not used if *analysisMode* is *eAnalysisModeMinimal*.

  Stage: Analyze

  The default value is *ccOCRDefs::kDefaultFontPitchType*.

  Throws

  ccOCRDefs::BadParams
  t is not a valid enum value.

minPitch

```
c_Int32 minPitch() const;
void minPitch(c_Int32 m);
```

- `c_Int32 minPitch() const;`

  Gets the minimum pitch, in pixels, that can occur between two characters, where the pitch is computed as specified by the pitch metric.

  Throws

  ccOCRDefs::BadParams
  $m < 0$.

- `void minPitch(c_Int32 m);`

  Sets the minimum pitch, in pixels, that can occur between two characters, where the pitch is computed as specified by the pitch metric.

  If the pitch between two fragments is smaller than this, then they must be considered to be part of the same character, unless the combined character would be too wide (as specified by *characterMaxWidth* and/or *characterMinAspect*).

  Parameters

  m The minimum pitch.

  Notes

  This parameter is not used if *analysisMode* is *eAnalysisModeMinimal*.

  Stage: Analyze

  The default value is 0.

  Throws

  ccOCRDefs::BadParams
  m < 0.

spaceParams

```
const ccOCCharSegmentSpaceParams& spaceParams() const;

void spaceParams(
  const ccOCCharSegmentSpaceParams& params);
```

- `const ccOCCharSegmentSpaceParams& spaceParams() const;`

  Gets parameters to control whether and how to insert space characters.

- `void spaceParams(`
  `const ccOCCharSegmentSpaceParams& params);`

  Sets parameters to control whether and how to insert space characters.

  Parameters

  params Parameters to control whether and how to insert space characters.

  Notes

  Stage: Analyze

Operators

operator==

```
bool operator==(const ccOCCharSegmentRunParams& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharSegmentRunParams& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCCharSegmentRunParams

ccOCCharSegmentSpaceParams

```
#include <ch_cvl/ocsegmnt.h>

class ccOCCharSegmentSpaceParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains parameters to control how space characters are handled by the OCR Segmenter tool.

Constructors/Destructors

ccOCCharSegmentSpaceParams

```
ccOCCharSegmentSpaceParams();
```

Constructor.

Notes

See individual getters for default values.

Compiler-generated copy constructor, assignment operator, and destructor are used.

Enumerations

SpaceInsertMode

```
enum SpaceInsertMode;
```

This enumeration defines how to handle insertion of space characters into gaps between other characters.

Value	Meaning
eSpaceInsertModeNone = 0	Never insert a space character, no matter how large an intercharacter gap is.
eSpaceInsertModeSingle = 1	Insert at most one space character per intercharacter gap, no matter how large the gap is.

Value	Meaning
eSpaceInsertModeMultiple = 2	Insert a number of space characters (zero or more) per intercharacter gap based on how large the gap is.
kDefaultSpaceInsertMode = *eSpaceInsertModeNone*	The default mode for inserting space characters.

SpaceScoreMode

```
enum SpaceScoreMode;
```

This enumeration defines how to compute the score of a space character.

Value	Meaning
eSpaceScoreModeOne = 0	Space characters always get a score of 1.0.
eSpaceScoreModeClutter = 1	The score of a space character is based on the fraction of pixels that are background. A space character that consists entirely of background receives a score of 1.0.
kDefaultSpaceScoreMode = *eSpaceScoreModeClutter*	The default mode for scoring a space character.

Public Member Functions

reset

```
void reset();
```

Resets all parameters to default values.

Notes

See individual getters for default values.

spaceInsertMode

```
SpaceInsertMode spaceInsertMode() const;

void spaceInsertMode(SpaceInsertMode mode);
```

- `SpaceInsertMode spaceInsertMode() const;`

  Gets the insert mode, which specifies how to handle insertion of space characters into gaps between other characters.

- `void spaceInsertMode(SpaceInsertMode mode);`

  Sets the insert mode, which specifies how to handle insertion of space characters into gaps between other characters.

  Parameters
  mode The insert mode.

  Throws
  ccOCRDefs::BadParams
  mode is not a valid enum value.

  The default value is *kDefaultSpaceInsertMode*.

spaceScoreMode

```
SpaceScoreMode spaceScoreMode() const;
void spaceScoreMode(SpaceScoreMode mode);
```

- `SpaceScoreMode spaceScoreMode() const;`

  Gets the score mode, which specifies how to handle scoring of space characters.

- `void spaceScoreMode(SpaceScoreMode mode);`

  Sets the score mode, which specifies how to handle scoring of space characters.

  Parameters
  mode The score mode.

  Throws
  ccOCRDefs::BadParams
  mode is not a valid enum value.

  The default value is *kDefaultSpaceScoreMode*.

spaceMinWidth

```
c_Int32 spaceMinWidth() const;
void spaceMinWidth(c_Int32 w);
```

- `c_Int32 spaceMinWidth() const;`

  Gets the minimum width of a space character, in pixels.

- `void spaceMinWidth(c_Int32 w);`

  Sets the minimum width of a space character, in pixels.

  Parameters

  w The minimum width of a space character.

  Throws

  ccOCRDefs::BadParams
  The width is < 1

  The default value is 10.

spaceMaxWidth

```
c_Int32 spaceMaxWidth() const;
void spaceMaxWidth(c_Int32 w);
```

- `c_Int32 spaceMaxWidth() const;`

  Gets the maximum width of a space character, in pixels.

- `void spaceMaxWidth(c_Int32 w);`

  Sets the maximum width of a space character, in pixels.

  Parameters

  w The maximum width of a space character.

  Throws

  ccOCRDefs::BadParams
  The width is < 1

  The default value is 100.

Operators

operator==

```
bool operator==(const ccOCCharSegmentSpaceParams& rhs)
const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCCharSegmentSpaceParams& rhs)
  const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for inequality.

ccOCCharSegmentSpaceParams

ccOCFont

```
#include <ch_cvl/ocfont.h>

class ccOCFont;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCFont** class represents a set of characters, including their character codes, images, and other metadata, primarily for use with OCR and OCV and also more generally to represent a font.

Constructors/Destructors

The default constructor, copy constructor, assignment operator, and destructor are used.

Public Member Functions

name32

```
const ccCvlString32& name32() const;

void name32(const ccCvlString32& n,
  TCHAR replacementIfUnrepresentableCharacterCode =
  _T('#'));
```

- ```
  const ccCvlString32& name32() const;
  ```

  Gets the name of this font, in Unicode UTF-32.

- ```
  void name32(const ccCvlString32& n,
    TCHAR replacementIfUnrepresentableCharacterCode =
    _T('#'));
  ```

  Sets the name of this font, in Unicode UTF-32.

  For the setter, if the name contains any 32-bit character codes that are not representable as TCHAR, those character codes are replaced by the specified replacement value in **name()**, but is kept in **name32()**.

Parameters

n The name of this font.

replacementIfUnrepresentableCharacterCode
Replacement value.

Notes

name32() and **name()** are always kept in sync, so that setting one also changes the other one appropriately.

The default value is an empty string.

name

```
const ccCvlString& name() const;

void name(const ccCvlString& n);
```

- `const ccCvlString& name() const;`

Gets the name of this font.

Throws

If any character in the underlying UTF-32 representation of the name cannot be represented as a TCHAR, it is replaced by a different value that was specified when the **name32()** setter was called.

- `void name(const ccCvlString& n);`

Sets the name of this font.

Parameters

n The name of this font.

Notes

name32() and **name()** are always kept in sync, so that setting one also changes the other one appropriately.

The default value is an empty string.

characters

```
const cmStd vector<ccOCChar>& characters() const;

void characters(
  const cmStd vector<ccOCChar>& c,
  bool makeUniqueInstanceKeys = false);
```

- `const cmStd vector<ccOCChar>& characters() const;`

  Gets the characters in this font.

- ```
  void characters(
    const cmStd vector<ccOCChar>& c,
    bool makeUniqueInstanceKeys = false);
  ```

  Sets the characters in this font.

  If *makeUniqueInstanceKeys* is true, then characters are all assigned unique keys; specifically, later characters in the set are reassigned unique instance values.

  Parameters

  c The characters in this font.

  makeUniqueInstanceKeys
  Whether characters are all assigned unique keys.

  Throws

  ccOCDefs::SameKey
  Two or more characters have the same key and *makeUniqueInstanceKeys* is false.

  ccOCDefs::SameKey
  Any of the characters' keys' instances are *ccOCCharKey::eUnspecified* and *makeUniqueInstanceKeys* is false.

  The default value is an empty vector.

hasCharacter

```
bool hasCharacter(const ccOCCharKey& key) const;
```

Returns true if a character with the specified key exists in this font.

Parameters

key The specified character key.

character

```
const ccOCChar& character(const ccOCCharKey& key) const;
```

Returns the character with the specified key.
```

Parameters
key The specified character key.

Throws
ccOCDefs::BadKey
No character has the specified character key.

add

```
void add(const ccOCChar& character);
```

Adds the character to the font.

Parameters
character The character to be added.

Throws
ccOCDefs::SameKey
Two or more characters have the same key, or
the character's key's instance is *ccOCCharKey::eUnspecified*.

remove

```
void remove(const ccOCCharKey& key);
void remove(const cmStd vector<ccOCCharKey>& keys);
```

- `void remove(const ccOCCharKey& key);`

Removes the character with the specified key from the font.

Parameters
key The specified character key.

Throws
ccOCDefs::BadKey
No character has the specified key.

- `void remove(const cmStd vector<ccOCCharKey>& keys);`

Removes the characters with the specified keys from the font.

Parameters
keys The specified character keys.

Throws
ccOCDefs::BadKey
No character has each specified key.

Notes

This function has no effect if **keys.size()** == 0.

removeAll

```
void removeAll();
```

Removes all characters from the font.

leading

```
cc2Vect leading() const;
void leading(const cc2Vect&);
```

- `cc2Vect leading() const;`

  Gets the leading for this font.

- `void leading(const cc2Vect&);`

  Sets the leading for this font.

  The default value is **cc2Vect(***0., 0.***)**

  Parameters

  cc2Vect The leading for this font.

encloseCellRect

```
ccRect encloseCellRect() const;
```

Gets the enclosing cell rectangle for the font, which encloses all of the individual character cell rectangles.

Notes

This function only considers the *eCellRectClient rectangles* for which *isRectSpecified* is true.

If no characters specify the *eCellRectClient* rectangles, this function returns a default constructed **ccRect**.

encloseMarkRect

```
ccRect encloseMarkRect() const;
```

Gets the enclosing mark rectangle for the font, which encloses all of the individual character mark rectangles.

Notes

This function only considers the *eMarkRectClient* rectangles for which *isRectSpecified* is true.

If no characters specify the *eMarkRectClient* rectangles, this function returns a default constructed **ccRect**.

assignImageFont

```
void assignImageFont(const ccImageFont& imageFont);
```

Resets this font to be essentially identical to the image font.

Parameters

imageFont The image font.

save

```
void save(const ccCvlString& filename) const;
```

Saves the font to a file with the given filename.

Parameters

filename The filename.

Notes

The standard file extension for use with **ccOCFont** font files is ".ocr".

load

```
void load(const ccCvlString& filename);
```

Loads the font from the file with the given filename; the file may be in ".ocr" format or in the older ".ocf" or ".ocm" formats.

The font is unchanged if the load fails.

Parameters

filename The filename.

Throws

ccOCDefs::BadFileFormat
The file does not contain a font in a supported format.

Notes

It can also throw other exceptions (for example, *ccArchive::Eof*, *ccArchive::UnknownVersion*, and so on). See file *ch_cvl/archive.h*.

tryLoad

```
bool tryLoad(const ccCvlString& filename);
```

Tries to load the font from the file with the given filename; the file may be in ".ocr" format or in the older ".ocf" or ".ocm" formats.

It returns true if successful or false if the file was in a bad format. Does not throw *ccOCDefs::BadFileFormat* if the file is in a bad format.

The font is unchanged if the load fails.

Parameters

filename The filename.

makeUniqueInstanceKey

```
ccOCCharKey makeUniqueInstanceKey(const ccOCCharKey& key)
const;
```

Returns a unique key to be used for a new instance of the given key, with the same Unicode character value.

Parameters

key The given key.

Operators

operator==

```
bool operator==(const ccOCFont& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCFont& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

ccOCFont

ccOCKeySet

```
#include <ch_cvl/oc.h>

class ccOCKeySet;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class comprises the set of keys for characters from one alphabet that may be used at a particular position within a **ccOCLine** object. It also maintains a set of indices for multiple characters that are allowed to appear at a line position. The OCV tool uses these indices to determine the characters to compare with the run-time character image found at a line position.

The OC Key Set class contains information about the keys of character models that is relevant to a particular position within a **ccOCLine** object. It contains the following information:

Keys

The set of unique keys for characters that may be used at a position. For example, if only numbers are possible for a position, this set would contain all of the number keys and none no letter keys.

Current keys

A subset of keys (represented as indices into keys) for the characters of which exactly one is currently expected at the position. That is, any one of the characters represented by the current keys is allowed at the position. Typically, the current keys simply represent character models for multiple versions (appearances) of the same character. In the case that one or more of the current character indices is -1, the current key is by definition unknown. That is, it could be a key for any character in any alphabet, not just the ones listed in the set (in the case of OCV, this would correspond to a wild card character).

Confusion keys overrides

Specified for any subset of keys in the key set. An override for a particular key specifies the set of keys for the characters that are *confusable* with the character for that particular key. If no override is specified for a particular key in the key set, the confusable keys for that key are simply the subset of keys in the alphabet (associated with the OC Line within which the position resides) for which **areConfusable()** is true and that are not part of the current key set.

Constructors/Destructors

ccOCKeySet

```
ccOCKeySet();

ccOCKeySet(const cmStd vector<c_Int32>& theKeys,
  c_Int32 currentKeyIndex = -1);

ccOCKeySet(const cmStd vector<c_Int32>& theKeys,
  const cmStd vector<c_Int32> curKeyIndices);

ccOCKeySet(const ccOCAlphabetPtrh_const& alphabet,
  const cmStd vector<ccCvlString> &charNames,
  const ccCvlString &currentCharName);
```

- `ccOCKeySet();`

  Constructs an empty set of keys and sets the current key index to -1. A current key index of -1 indicates a wildcard for the position. Anything found at the position will pass verification.

- `ccOCKeySet(const cmStd vector<c_Int32>& theKeys, c_Int32 currentKeyIndex = -1);`

  Constructs a key set using the supplied keys and current key index.

  Parameters

  theKeys The keys of character models from one alphabet from which to create this key set.

  currentKeyIndex The index of the key to set as the current key or -1 to specify that the current key corresponds to a wildcard character. Setting *currentKeyIndex* to -1 means that anything found at the position will pass verification.

  Throws

  ccOCDefs::SameKey
  Two or more keys are the same. Keys must be unique.

  ccOCDefs::BadIndex
  The current key index does not exist in the key set and is not -1.

- ```
  ccOCKeySet(const cmStd vector<c_Int32>& theKeys,
    const cmStd vector<c_Int32> curKeyIndices);
  ```

  Constructs a key set using the given keys and a set of current key indices. You can supply multiple versions of a character to be verified at a line position by selecting these from the key set. At run-time, the OCV tool determines which of the current characters is the best match for the found character.

  Parameters

  theKeys The keys of character models from one alphabet from which to create this key set.

  curKeyIndices A set of indices from the *theKeys* for characters that may appear at a line position and for which you want to perform verification.

  If you set an index value to -1, the OCV tool treats the set as a single wildcard character, and therefore any character found at the line position passes verification.

  Throws

  ccOCDefs::SameKey
  Two or more keys are the same. Keys must be unique.

  ccOCDefs::BadIndex
  At least one of the current key indices is not valid or unique, and is not -1.

- ```
  ccOCKeySet(const ccOCAlphabetPtrh_const& alphabet,
    const cmStd vector<ccCvlString> &charNames,
    const ccCvlString &currentCharName);
  ```

  Constructs a key set using the given alphabet, character names, and current character name. All characters from the alphabet with the same name as *currentCharName* are marked as current keys: these may, for example, include multiple versions of a character. At run-time, the OCV tool determines which of the current characters is the best match for the character found at a line position.

  Parameters

  alphabet The alphabet that contains characters from which to create the key set.

  charNames The names of characters in the alphabet to include in the key set.

  currentCharName
  The character name used to determine which members of the key set are selected as current character keys.

Throws

ccOCDefs::SameKey
Two or more character names are the same.

ccOCDefs::BadKey
A character in *charNames* cannot be found in the specified alphabet.

ccOCDefs::BadIndex
The current character is not valid for the specified set of possible character names.

Operators

operator==

```
bool operator==(const ccOCKeySet& other) const;
```

Returns true if this key set is equal to the specified key set.

Parameters

other The key set with which to compare for equality.

operator!=

```
bool operator!=(const ccOCKeySet& other) const;
```

Returns true if this key set is not equal to the specified key set.

Parameters

other The key set with which to compare for inequality.

Public Member Functions

keys

```
void keys(const cmStd vector<c_Int32>& ks);
const cmStd vector<c_Int32>& keys() const;
```

- ```
  void keys(const cmStd vector<c_Int32>& ks);
  ```

Sets the keys for this key set. Any existing confusion override keys are discarded when this function is called.

Parameters

ks The keys of character models from one alphabet.

Throws

ccOCDefs::SameKey
Two or more keys are the same. Keys must be unique.
  ```

ccOCDefs::BadIndex
The current key index does not exist in the key set and is not -1.

- `const cmStd vector<c_Int32>& keys() const;`

  Returns the keys for this key set.

currentKeyIndex

```
void currentKeyIndex(c_Int32 index);

c_Int32 currentKeyIndex() const;
```

- `void currentKeyIndex(c_Int32 index);`

  Sets the index of the current key for the key set (if the set of current key indices has only one element). For more flexibility in setting current keys, use **currentKeyIndices()**.

  The OCV tool uses the current key index to determine the character to compare with the run-time character image found at a line position. A current key index of -1 indicates a wildcard, and anything found at the position will pass verification.

  Parameters

  index The index of the key to set as the current key or -1 to specify that the current key corresponds to a wildcard character.

  Throws

  ccOCDefs::BadIndex
  The supplied key index does not exist in the key set and is not -1.

  ccOCDefs::BadParam
  The set of current key indices has more than one element.

- `c_Int32 currentKeyIndex() const;`

  Returns the index of the current key for the key set (if the set of current key indices has only one element).

  Throws

  ccOCDefs::BadParam
  The set of current key indices has more than one element.

currentKeyIndices

```
void currentKeyIndices(
  const cmStd vector<c_Int32> &indices);

const cmStd vector<c_Int32> &currentKeyIndices() const;
```

- ```
  void currentKeyIndices(
    const cmStd vector<c_Int32> &indices);
  ```

  Sets the indices of the current keys for the key set.

  The OCV tool uses the current key indices to determine the character to compare with the run-time character image found at a line position. To specify a wildcard character for a line position, set one or more of the current key indices to -1. Any character found at the position will pass verification.

  Parameters

  indices The set of character key indices to use as the current keys. If any any index value is -1, the set is treated as a single wildcard character.

  Throws

  ccOCDefs::BadIndex
  A supplied key index is not unique or valid, and it is not -1.

- ```
  const cmStd vector<c_Int32> &currentKeyIndices() const;
  ```

  Returns the indices of the current keys for the key set.

isCurrentWildcard

```
bool isCurrentWildcard() const;
```

Returns true if any of the current key indices is set to -1.

currentKey

```
c_Int32 currentKey() const;
```

Returns the current key if the set of current key indices has only one element.

Throws

ccOCDefs::BadKey
The current key index is set to -1, the wildcard character.

ccOCDefs::BadParams
The set of current key indices has more than one element.

currentKeys

```
cmStd vector<c_Int32> currentKeys() const;
```

Returns the current keys.

Throws

ccOCDefs::BadKey
isCurrentWildcard() returns true.

confusionOverrides

```
void confusionOverrides(c_Int32 key,
 const cmStd vector<c_Int32>& confusionKeys);

const cmStd vector<c_Int32>& confusionOverrides(
 c_Int32 key) const;
```

- ```
  void confusionOverrides(c_Int32 key,
   const cmStd vector<c_Int32>& confusionKeys);
  ```

  Sets the confusion override keys for the specified key.

  If confusion override keys are defined for a particular key, then the OCV tool will use the override keys as the confusable keys for the particular key. If no override keys are defined, then all characters in the current alphabet that have confusion scores above the confusion threshold (and which are not members of the current key set for the character position) are treated as confusable.

  Specifying override keys lets you perform OCR (Optical Character Recognition) using the OCV tool. By specifying which characters are confusable, you can force the tool to check for confusion among the characters defined as part of the current key set. This lets you determine not only that one of the valid characters is present, but which one is present.

  Notes

  Any confusion keys that are not valid in the alphabet associated with the line in which this key is used are ignored. Duplicate confusion keys are counted only once.

  Parameters

  key The key for which the overrides are defined.

  confusionKeys The override keys.

  Throws

  ccOCDefs::BadKey
  key is not a member of this **ccOCKeySet**.
  ```

- `const cmStd vector<c_Int32>& confusionOverrides( c_Int32 key) const;`

  Returns the confusion override keys for the specified key.

  Parameters

  key The key for which to return the override keys.

  Throws

  ccOCDefs::BadKey

  key is not a member of this **ccOCKeySet** or there are no confusion overrides defined for *key*.

removeConfusionOverrides

```
void removeConfusionOverrides(c_Int32 key);
```

Removes all the confusion overrides for the specified key.

Throws

ccOCDefs::BadKey

key is not a member of this **ccOCKeySet** or there are no confusion overrides defined for *key*.

hasConfusionOverrides

```
bool hasConfusionOverrides(c_Int32 key);
```

Returns true if this key has confusion overrides defined for it, false otherwise.

Throws

ccOCDefs::BadKey

key is not a member of this **ccOCKeySet**.

ccOCLine

```
#include <ch_cvl/oc.h>

class ccOCLine : public ccPersistent, public ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class represents a spatial arrangement of characters from one alphabet. A typical line consists of evenly spaced, upright characters as found on a product label, however you can configure arbitrary arrangements of characters by supplying information about the uncertainty in position, angle, and scale of each character. For each line position, this class maintains a set of keys for characters that can appear at the position. An optical verification tool uses the information provided by the **ccOCLine** class to verify characters in a line of text.

Constructors/Destructors

ccOCLine

```
ccOCLine();

ccOCLine(const cmStd vector<ccOCKeySet>& keySetSequence,
  const ccOCAlphabetPtrh_const& alphabet,
  const cc2Rigid& spacing, double charScale = 1.0,
  double transUnc = 0.0,
  const ccRadian& rotUnc = ccRadian(0),
  double scaleUnc = 0.0,
  const ccCvlString& name = cmT(""));

ccOCLine(const cmStd vector<ccOCKeySet>& keySetSequence,
  const ccOCAlphabetPtrh_const& alphabet,
  const cmStd vector<cc2Rigid>& charPoses,
  double charScale = 1.0,
  double transUnc = 0.0,
  const ccRadian & rotUnc = ccRadian(0),
  double scaleUnc = 0.0,
  const ccCvlString& name = cmT(""));

ccOCLine(const cmStd vector<ccOCKeySet>& keySetSequence,
  const ccOCAlphabetPtrh_const& alphabet,
  const cmStd vector<cc2Rigid>& charPoses,
  const cmStd vector<double>& charScales,
```

```
  const cmStd vector<double>& transUncs,
  const cmStd vector<ccRadian>& rotUncs,
  const cmStd vector<double>& scaleUncs,
  const ccCvlString& name = cmT(""));
```

- `ccOCLine();`

  Constructs an empty line with an empty alphabet.

- ```
  ccOCLine(const cmStd vector<ccOCKeySet>& keySetSequence,
    const ccOCAlphabetPtrh_const& alphabet,
    const cc2Rigid& spacing,
    double charScale = 1.0,
    double transUnc = 0.0,
    const ccRadian& rotUnc = ccRadian(0),
    double scaleUnc = 0.0,
    const ccCvlString& name = cmT(""));
  ```

  Constructs a line using the specified alphabet and sequence of character key sets. Characters are uniformly spaced with respect to their origins, and the scaling and uncertainty factors apply to all characters in the line.

  Parameters

  keySetSequence A sequence of key sets that indicates which characters can appear at each position in the line. Specify the key sets in line position order.

  alphabet The alphabet to associate with this line. The alphabet must already be compiled.

  spacing The spacing between the origin of each character. This factor is applied to all characters in the line.

  charScale The scaling for all characters. This must be greater than 0.

  transUnc The translation uncertainty for all characters. This must be greater than or equal to 0.

  rotUnc The rotation uncertainty for all characters. This must be greater than or equal to 0.

  scaleUnc The scaling uncertainty for all characters. This must be greater than or equal to 0.

  name The name of this line.
  ```

Notes

For the typical case of a horizontal line of upright characters (x- and y-axes of all characters parallel with the x- and y-axes of the first character in the line), specify a *spacing* value that has only an x-axis translation component.

Throws

ccOCDefs::BadParams

If *charScale* is less than or equal to 0,
or if a value for *transUncs*, *rotUncs*, or *scaleUncs* is less than 0,
or if *alphabet* is NULL.

ccOCDefs::BadKey

A character model associated with a key does not exist in the specified alphabet.

- ```
  ccOCLine(const cmStd vector<ccOCKeySet>& keySetSequence,
    const ccOCAlphabetPtrh_const& alphabet,
    const cmStd vector<cc2Rigid>& charPoses,
    double charScale = 1.0,
    double transUnc = 0.0,
    const ccRadian & rotUnc = ccRadian(0),
    double scaleUnc = 0.0,
    const ccCvlString& name = cmT(""));
  ```

  Constructs a line using the specified alphabet and sequence of character key sets. Each character has a rigid pose with respect to an implicit character line coordinate system. Scaling and uncertainty factors apply to all characters.

  Parameters

  keySetSequence A sequence of key sets that indicates which characters can appear at each position in the line. Specify the key sets in line position order.

  alphabet The alphabet to associate with this line. The alphabet must already be compiled.

  charPoses The rigid pose of each character, specified in line position order.

  charScale The scaling for all characters. This must be greater than 0.

  transUnc The translation uncertainty for all characters. This must be greater than or equal to 0.

  rotUnc The rotation uncertainty for all characters. This must be greater than or equal to 0.

  scaleUnc The scaling uncertainty for all characters. This must be greater than or equal to 0.
  ```

name The name of this line.

Throws

ccOCDefs::BadParams
If the number of elements in *keySetSequence* does not equal the number of character poses,
or if *charScale* is less than or equal to 0,
or if a value for *transUncs*, *rotUncs*, or *scaleUncs* is less than 0,
or if *alphabet* is NULL.

ccOCDefs::BadKey
A character model associated with a key does not exist in the specified alphabet.

- ```
  ccOCLine(const cmStd vector<ccOCKeySet>& keySetSequence,
    const ccOCAlphabetPtrh_const& alphabet,
    const cmStd vector<cc2Rigid>& charPoses,
    const cmStd vector<double>& charScales,
    const cmStd vector<double>& transUncs,
    const cmStd vector<ccRadian>& rotUncs,
    const cmStd vector<double>& scaleUncs,
    const ccCvlString& name = cmT(""));
  ```

Constructs a line using the specified sequence of character key sets and an alphabet. Each character has a rigid pose with respect to an implicit line coordinate system. You supply scaling and uncertainty factors for each character in the line.

Parameters

keySetSequence A sequence of key sets that indicates which characters can appear at each position in the line. Specify the key sets in line position order.

alphabet The alphabet to associate with this line. The alphabet must already be compiled.

charPoses The pose for each character, specified in line position order.

charScales The scale for each character, specified in line position order. Each of these values must be greater than 0.

transUncs The translation uncertainty for each character, specified in line position order. These values must be greater than or equal to 0.

rotUncs The rotation uncertainty for each character, specified in line position order. These values must be greater than or equal to 0.

scaleUncs The scaling uncertainty for each character, specified in line position order. These values must be greater than or equal to 0.
  ```

name The name of the line.

Throws

ccOCDefs::BadParams
If the number of elements in each of the supplied vectors are unequal,
or if *charScale is less than or equal to 0,*
or if a value for *transUncs*, *rotUncs*, or *scaleUncs* is less than 0,
or if alphabet is NULL.

ccOCDefs::BadKey
A character model associated with a key does not exist in the specified alphabet.

Notes

This constructor allows you to specify different scales for each character in a line. However, varying character scales decreases the ability of an optical tool to perform effective character recognition and verification. This is because confusion scores are based on comparing characters at similar positions angles, and scales. You should try to use separate lines to describe characters at different scales.

Operators

operator==

```
bool operator==(const ccOCLine& other) const;
```

Return true if this line is equal to the specified line.

Parameters

other The line with which to compare for equality.

Notes

The equality operator considers the alphabets associated with two lines equivalent if their names are the same. The equality operator does not compare the models within the alphabets for equality.

operator!=

```
bool operator!=(const ccOCLine& other) const;
```

Returns true if this line is not equal to the specified line.

Parameters

other The line with which to compare for inequality.

Public Member Functions

numPositions

```
c_Int32 numPositions() const;
```

Returns the number of character positions in this line.

numNormalChars

```
c_Int32 numNormalChars() const;
```

Returns the number of normal characters in this line.

isWildcard

```
bool isWildcard(c_Int32 posIndex) const;
```

Returns true if the current character at the supplied index is a wildcard character.

Parameters

posIndex The index of the character position to test.

Throws

ccOCDefs::BadIndex
posIndex value is not a valid position index.

character

```
const ccOCModelPtrh_const & character(
  c_Int32 posIndex) const;
```

Returns the first character model for the current characters at the specified position.

Parameters

posIndex The character position index.

Throws

ccOCDefs::BadIndex
The position index is invalid or one of the current characters at this position is a wildcard character.

characters

```
cmStd vector<ccOCModelPtrh_const> characters(
  c_Int32 posIndex) const;
```

Returns the character models for the current characters at the specified position.

Parameters

posIndex The character position index.

Throws

ccOCDefs::BadIndex

The position index is invalid or one of the current characters at this position is a wildcard character.

alphabet

```
const ccOCAlphabetPtrh_const& alphabet() const;
```

Returns the alphabet for this line.

keySetSequence

```
const cmStd vector<ccOCKeySet>& keySetSequence() const;
```

Returns the character key set sequence for this line.

charPoses

```
const cmStd vector<cc2Rigid>& charPoses() const;
```

Returns the character poses for this line with respect to the line coordinate system.

charScales

```
const cmStd vector<double>& charScales() const;
```

Returns the character scales for this line with respect to the line coordinate system.

translationUncertainties

```
const cmStd vector<double>&
  translationUncertainties() const;
```

Returns the character translation uncertainties.

rotationUncertainties

```
const cmStd vector<ccRadian>&
  rotationUncertainties() const;
```

Returns the character rotation uncertainties.

scaleUncertainties

```
const cmStd vector<double>& scaleUncertainties() const;
```

Returns the character scale uncertainties.

name

```
const ccCvlString& name() const;
```

Returns the name of this line.

encloseRect

```
ccRect encloseRect(
  const cc2Xform& clientFromLine = cc2Xform(),
  const cc2Xform& lineOffset = cc2Xform(),
  const cc2Xform& clientFromImage = cc2Xform(),
  double translationUncertainty = 0,
  const ccRadian& rotationUncertainty = ccRadian(0),
  double scaleUncertainty = 0,
  bool includeConfusion = false) const;
```

Returns the smallest rectangle that contains this line. The enclosing rectangle is aligned with respect to the specified image coordinate system.

Parameters

clientFromLine

A transformation that maps the line's coordinate system to the client coordinate system.

lineOffset

A transformation that maps the origin of the line to the origin of the client coordinate system.

clientFromImage

A transformation that maps the image coordinate system to the client coordinate system.

translationUncertainty

The translation uncertainty for the line. This value must be greater than or equal to 0.

rotationUncertainty

The rotation uncertainty for the line. This value must be greater than or equal to 0.

scaleUncertainty

The scale uncertainty for the line. This value must be greater than or equal to 0. You should not specify a value greater than 0.10 for the scale uncertainty.

includeConfusion

If true, the returned enclosing rectangle contains any confusing characters in place of those nominally present.

Throws

ccOCDefs::BadParams

translationUncertainty, *rotationUncertainty*, or *scaleUncertainty* is less than 0.

Typedefs

ccOCLinePtrh

```
typedef ccPtrHandle<ccOCLine> ccOCLinePtrh;
```

ccOCLinePtrh_const

```
typedef ccPtrHandle_const<ccOCLine> ccOCLinePtrh_const;
```

ccOCLine

ccOCLineArrangement

```
#include <ch_cvl/oc.h>

class ccOCLineArrangement : public ccPersistent, public ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class represents a spatial arrangement of lines. A typical line arrangement consists of horizontal, parallel lines as found on a product label, however you can configure arbitrary arrangements of lines by supplying information about the uncertainty in position, angle, and scale for each line. An optical verification tool uses the information provided by this class to verify the lines in an area of text.

Constructors/Destructors

ccOCLineArrangement

```
ccOCLineArrangement();

ccOCLineArrangement(
  const cmStd vector<ccOCLinePtrh_const>& lines,
  const cc2Rigid& spacing,
  double transUnc = 0.0,
  const ccRadian& rotUnc = ccRadian(0),
  const ccCvlString& name = cmT(""));

ccOCLineArrangement(
  const cmStd vector<ccOCLinePtrh>& lines,
  const cc2Rigid& spacing,
  double transUnc = 0.0,
  const ccRadian& rotUnc = ccRadian(0),
  const ccCvlString& name = cmT(""));

ccOCLineArrangement(
  const cmStd vector<ccOCLinePtrh_const>& lines,
  const cmStd vector<cc2Rigid>& linePoses,
```

```
  double transUnc = 0.0,
  const ccRadian & rotUnc = ccRadian(0),
  const ccCvlString& name = cmT(""));

ccOCLineArrangement(
  const cmStd vector<ccOCLinePtrh>& lines,
  const cmStd vector<cc2Rigid>& linePoses,
  double transUnc = 0.0,
  const ccRadian & rotUnc = ccRadian(0),
  const ccCvlString& name = cmT(""));

ccOCLineArrangement(
  const cmStd vector<ccOCLinePtrh_const>& lines,
  const cmStd vector<cc2Rigid>& linePoses,
  const cmStd vector<double>& transUncs,
  const cmStd vector<ccRadian>& rotUncs,
  const ccCvlString& name = cmT(""));

ccOCLineArrangement(
  const cmStd vector<ccOCLinePtrh>& lines,
  const cmStd vector<cc2Rigid>& linePoses,
  const cmStd vector<double>& transUncs,
  const cmStd vector<ccRadian>& rotUncs,
  const ccCvlString& name = cmT(""));
```

- `ccOCLineArrangement();`

  Constructs a line arrangement with no lines.

- ```
  ccOCLineArrangement(
    const cmStd vector<ccOCLinePtrh_const>& lines,
    const cc2Rigid& spacing,
    double transUnc = 0.0,
    const ccRadian& rotUnc = ccRadian(0),
    const ccCvlString& name = cmT(""));
  ```

  Constructs a line arrangement from the specified set of lines. Lines are uniformly spaced with respect their origins, and the scaling and uncertainty factors apply to all characters in the line.

  Parameters

lines	The set of lines that this arrangement comprises. If any line contains space characters in all positions, the line arrangement is invalid.
spacing	The spacing between the origin of each line. This factor is applied to all lines in the arrangement.
  ```

transUnc The translation uncertainty for all lines. This must be greater than or equal to 0.

rotUnc The rotation uncertainty for all lines. This must be greater than or equal to 0.

name The name of this line arrangement

Notes

For the typical case of parallel, horizontal lines of text, specify a *spacing* value that has only a y-axis translation component.

Throws

ccOCDefs::BadParams

transUnc or *rotUnc* is less than 0.

- ```
  ccOCLineArrangement(
    const cmStd vector<ccOCLinePtrh>& lines,
    const cc2Rigid& spacing,
    double transUnc = 0.0,
    const ccRadian& rotUnc = ccRadian(0),
    const ccCvlString& name = cmT(""));
  ```

Constructs a line arrangement from the specified set of lines. Lines are uniformly spaced with respect their origins, and the scaling and uncertainty factors apply to all characters in the line.

Parameters

lines The set of lines that this arrangement comprises. If any line contains space characters in all positions, the line arrangement is invalid.

spacing The spacing between the origin of each line. This factor is applied to all lines in the arrangement.

transUnc The translation uncertainty for all lines. This must be greater than or equal to 0.

rotUnc The rotation uncertainty for all lines. This must be greater than or equal to 0.

name The name of this line arrangement.

Notes

For the typical case of parallel, horizontal lines of text, specify a *spacing* value that has only a y-axis translation component.
  ```

Throws

ccOCDefs::BadParams

transUnc or *rotUnc* is less than 0.

- ```
  ccOCLineArrangement(
    const cmStd vector<ccOCLinePtrh_const>& lines,
    const cmStd vector<cc2Rigid>& linePoses,
    double transUnc = 0.0,
    const ccRadian & rotUnc = ccRadian(0),
    const ccCvlString& name = cmT(""));
  ```

  Constructs a line arrangement from the specified set of lines. For each line, you can supply an arbitrary pose with respect to the implicit line arrangement coordinate system. You can also supply translation and rotation uncertainty factors that apply to all lines in the arrangement.

  Parameters

  lines — The set of lines which this arrangement comprises. If any line contains space characters in all positions, the line arrangement is invalid.

  linePoses — The set of poses for the lines in the arrangement, specified in line order and with respect to the implicit line arrangement coordinate system.

  transUnc — The translation uncertainty for all lines. This must be greater than or equal to 0.

  rotUnc — The rotation uncertainty for all lines. This must be greater than or equal to 0.

  name — The name of this line arrangement.

  Throws

  ccOCDefs::BadParams

  The number of line poses does not equal the number of lines; or *transUnc* or *rotUnc* is less than 0.

- ```
  ccOCLineArrangement(
    const cmStd vector<ccOCLinePtrh>& lines,
    const cmStd vector<cc2Rigid>& linePoses,
  ```

```
double transUnc = 0.0,
const ccRadian & rotUnc = ccRadian(0),
const ccCvlString& name = cmT(""));
```

Constructs a line arrangement from the specified set of lines. For each line, you can supply an arbitrary pose with respect to the implicit line arrangement coordinate system. You can also supply translation and rotation uncertainty factors that apply to all lines in the arrangement.

Parameters

lines The set of lines that this arrangement comprises. If any line contains space characters in all positions, the line arrangement is invalid.

linePoses The set of poses for the lines in the arrangement, specified in line order and with respect to the implicit line arrangement coordinate system.

transUnc The translation uncertainty for all lines. This must be greater than or equal to 0.

rotUnc The rotation uncertainty for all lines. This must be greater than or equal to 0.

name The name of this line arrangement.

Throws

ccOCDefs::BadParams
The number of line poses does not equal the number of lines; or *transUnc* or *rotUnc* is less than 0.

- ```
  ccOCLineArrangement(
    const cmStd vector<ccOCLinePtrh_const>& lines,
    const cmStd vector<cc2Rigid>& linePoses,
    const cmStd vector<double>& transUncs,
    const cmStd vector<ccRadian>& rotUncs,
    const ccCvlString& name = cmT(""));
  ```

  Constructs a line arrangement from the specified set of lines. For each line, you can supply an arbitrary pose with respect to the implicit line arrangement coordinate system. You also supply the uncertainty in translation and rotation for each line.

  Parameters

  lines The set of lines that this arrangement comprises.

  linePoses The set of poses for the lines in the arrangement, specified in line order and with respect to the implicit line arrangement coordinate system.
  ```

transUncs The set of translation uncertainty factors for the lines, specified in line order. These values must be greater than or equal to 0.

rotUncs The set of rotation uncertainty factors for the lines, specified in line order. These values must be greater than or equal to 0.

name The name of this line arrangement.

Throws

ccOCDefs::BadParams
transUnc or *rotUnc* is less than 0.

- ```
  ccOCLineArrangement(
    const cmStd vector<ccOCLinePtrh>& lines,
    const cmStd vector<cc2Rigid>& linePoses,
    const cmStd vector<double>& transUncs,
    const cmStd vector<ccRadian>& rotUncs,
    const ccCvlString& name = cmT(""));
  ```

  Constructs a line arrangement from the specified set of lines. For each line, you can supply an arbitrary pose with respect to the implicit line arrangement coordinate system. You also supply the uncertainty in translation and rotation for each line.

  Parameters

  lines The set of lines that this arrangement comprises. If any line contains space characters in all positions, the line arrangement is invalid.

  linePoses The set of poses for the lines in the arrangement, specified in line order and with respect to the implicit line arrangement coordinate system.

  transUncs The set of translation uncertainty factors for the lines, specified in line order. These values must be greater than or equal to 0.

  rotUncs The set of rotation uncertainty factors for the lines, specified in line order. These values must be greater than or equal to 0.

  name The name of this line arrangement.

  Throws

  ccOCDefs::BadParams
  The number of line poses or uncertainties does not equal the number of lines; or *transUnc* or *rotUnc* is less than 0.
  ```

Operators

operator==

```
bool operator==(const ccOCLineArrangement& other) const;
```

Return true if this line arrangement is equal to the specified line arrangement.

Parameters

other The line arrangement with which to compare for equality.

operator!=

```
bool operator!=(const ccOCLineArrangement& other) const;
```

Return true if this line arrangement is not equal to the specified line arrangement.

Parameters

other The line arrangement with which to compare for inequality.

Public Member Functions

numLines

```
const c_Int32 numLines() const;
```

Returns the number of lines in this arrangement.

lines

```
const cmStd vector<ccOCLinePtrh_const>& lines() const;
```

Returns the set of lines for this arrangement.

linePoses

```
const cmStd vector<cc2Rigid>& linePoses() const;
```

Returns the set of line poses for this arrangement. Line poses are aligned with respect to the line arrangement coordinate system.

translationUncertainties

```
const cmStd vector<double>&
  translationUncertainties() const;
```

Returns the set of translation uncertainties for the lines in this arrangement.

rotationUncertainties

```
const cmStd vector<ccRadian>&
  rotationUncertainties() const;
```

Returns the set of rotation uncertainties for the lines in this arrangement.

name

```
const ccCvlString& name() const;
```

Returns the name of this line arrangement.

encloseRect

```
ccRect encloseRect(
  const cc2Xform& clientFromArr = cc2Xform(),
  const cc2Xform& arrOffset = cc2Xform(),
  const cc2Xform& clientFromImage = cc2Xform(),
  double translationUncertainty = 0,
  const ccRadian& rotationUncertainty = ccRadian(0),
  double scaleUncertainty = 0,
  bool includeConfusion = false) const;
```

Returns the smallest rectangle that contains this line arrangement. The enclosing rectangle is aligned with respect to the specified image coordinate system. If *includeConfusion* is true, the enclosing rectangle also contains any confusing characters in place of those nominally present.

Parameters

clientFromArr A transformation that maps the line arrangement's coordinate system to the client coordinate system.

arrOffset A transformation that maps the origin of the line arrangement to the origin of the client coordinate system.

clientFromImage A transformation that maps the image coordinate system to the client coordinate system.

translationUncertainty
The translation uncertainty for the line arrangement. This value must be greater than or equal to 0.

rotationUncertainty
The rotation uncertainty for the line arrangement. This value must be greater than or equal to 0.

scaleUncertainty
The scale uncertainty for the line arrangement. This value must be greater than or equal to 0. You should not specify a value greater than 0.10 for the scale uncertainty.

includeConfusion
If true, the returned enclosing rectangle contains any confusing characters in place of those nominally present.

Throws

ccOCDefs::BadParams

translationUncertainty, *rotationUncertainty*, or *scaleUncertainty* is less than 0.

Typedefs

ccOCLineArrangementPtrh

```
typedef ccPtrHandle<ccOCLineArrangement>
  ccOCLineArrangementPtrh;
```

ccOCLineArrangementPtrh_const

```
typedef ccPtrHandle_const<ccOCLineArrangement>
  ccOCLineArrangementPtrh_const;
```

ccOCLineArrangement

ccOCModel

```
#include <ch_cvl/oc.h>

class ccOCModel : public ccPersistent, public ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class represents a single character for optical character verification by the OCV tool.

Constructors/Destructors

ccOCModel

```
ccOCModel();

ccOCModel(c_Int32 key,
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPelRect& imageArea,
  CharType ctype,
  const ccCvlString& name = cmT(""),
  const ccCvlString& description = cmT(""));

ccOCModel(c_Int32 key,
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccPelRect& imageArea, CharType ctype,
  const ccCvlString& name = cmT(""),
  const ccCvlString& description = cmT(""));
```

- ```
  ccOCModel();
  ```

  The default constructor creates a model for a blank character with no area.

- ```
  ccOCModel(c_Int32 key,
    const ccPelBuffer_const<c_UInt8>& image,
    const ccPelRect& imageArea,
  ```

```
  CharType ctype,
  const ccCvlString& name = cmT(""),
  const ccCvlString& description = cmT(""));
```

Creates a character model from the specified parameters.

Parameters

key A unique key used to refer to the character model.

image The image for the character model.

imageArea The rectangle defining the area of the character in image coordinates.

ctype The character type, which can be either *eNormal*, a normal character, or *eBlank*, a blank character.

name A string used to name the character model

description A string used to describe the character model.

Throws

ccOCDefs::BadImage
The *image* is unbound but *ctype* is *eNormal*.

ccOCDefs::NotImplemented
The *image* is bound but *ctype* is *eBlank*.

ccOCDefs::BadParams
The *image* is bound and the *imageArea* does not fit within the window region.

-
```
ccOCModel(c_Int32 key,
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccPelRect& imageArea, CharType ctype,
  const ccCvlString& name = cmT(""),
  const ccCvlString& description = cmT(""));
```

Creates a character model that includes an image mask. If the mask is unbound, this constructor behaves the same as the previous overloaded constructor.

Parameters

key A unique key used to refer to the character model.

image The image for the character model.

mask An image that denotes certain areas of the character image that the OCV tool should ignore when training.

imageArea The rectangle defining the area of the character in image coordinates.

ctype The character type, which can be either *eNormal*, a normal character, or *eBlank*, a blank character.

name A string used to name the character model.

description A string used to describe the character model.

Throws

ccOCDefs::BadImage
The *image* is unbound but *ctype* is *eNormal*.

ccOCDefs::NotImplemented
The *image* is bound but *ctype* is *eBlank*.

ccOCDefs::BadParams
image and *mask* are both bound but their window regions are unequal; or the *image* or *mask* is bound but the specified image area does not fit within the window region.

Operators

operator==

```
bool operator==(const ccOCModel& other) const;
```

Returns true if this character model is equal to the specified character model.

Parameters

other A character model to compare with this one.

operator!=

```
bool operator!=(const ccOCModel& other) const;
```

Returns true if this object is not equal to the specified character model.

Parameters

other A character model to compare with this one.

Enumerations

CharType `enum CharType`

This enumeration defines the types of character models.

Value	Meaning
eNormal	Specifies a model for a normal character. A normal character can be verified by matching its shape. For example, any alphanumeric character or distinctive shape, such as a logo or graphic, is a normal character.
eBlank	Specifies a model for a blank character. A blank character represents “white space,” and can be verified by the absence of shape.

Public Member Functions

key

```
void key(c_Int32 key);
c_Int32 key() const;
```

- `void key(c_Int32 key);`

  Sets the key with which you refer to this character. Typically, you supply the ASCII code for the character as its key.

  Parameters

  key The character key.

- `c_Int32 key() const;`

  Returns the key for this character.

charType `CharType charType() const;`

Returns the character type for this model.

name

```
void name(const ccCvlString& n);
const ccCvlString& name() const;
```

- `void name(const ccCvlString& n);`

  Sets the name of this character model.

  Parameters

  n The name string.

- `const ccCvlString& name() const;`

  Returns the name for this character model.

description

```
void description(const ccCvlString& n);
const ccCvlString& description() const;
```

- `void description(const ccCvlString& n);`

  Sets the description string for this character model.

  Parameters

  n A string that describes the character model.

- `const ccCvlString& description() const;`

  Returns the description string for this character model.

image

```
void image(const ccPelBuffer_const<c_UInt8>& image);
const ccPelBuffer_const<c_UInt8>& image() const;
```

- `void image(const ccPelBuffer_const<c_UInt8>& image);`

  Sets the image for this character model. If the image is bound, it represents the prototype for a normal character. If unbound, the image represents a blank character. The image client coordinate system determines the origin, orientation, and scale of the character.

  Parameters

  image A pel buffer that contains either a bound or unbound image for the character.

Throws

ccOCDefs::BadImage
image is unbound but the character type for this model is *eNormal*.

ccOCDefs::NotImplemented
image is bound but the character type for this model is *eBlank*.

ccOCDefs::BadParams
image and the character's mask are both bound but their window regions are unequal; or the *image* is bound but its area does not fit within the window region.

- `const ccPelBuffer_const<c_UInt8>& image() const;`

Returns the image for this character model.

mask

```
void mask(const ccPelBuffer_const<c_UInt8>& mask);
const ccPelBuffer_const<c_UInt8>& mask() const;
```

- `void mask(const ccPelBuffer_const<c_UInt8>& mask);`

Sets the mask for this character model. A mask specifies certain areas of the character image that the OCV tool should ignore when training. Pixels with a value less than 128 denote Don't Care pixels. Those with a value equal to or greater than 128 denote Care pixels. An unbound mask is equivalent to a mask of all Care pixels (the same as specifying no mask).

Parameters

mask A pel buffer that contains the character mask.

Throws

ccOCDefs::BadParams
The *mask* and the character image are both bound but their window regions are unequal; or the *mask* is bound but its area does not fit within the window region.

- `const ccPelBuffer_const<c_UInt8>& mask() const;`

Returns the mask for this character model or, if you have not specified a mask, an unbound image buffer.

model

```
void model(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask);
```

Sets both the image and mask for this character model. If the mask is unbound, this function has the same effect as the **image()** function.

Parameters

image A pel buffer that contains either a bound or unbound image for the character.

mask A pel buffer that contains the character mask.

Throws

ccOCDefs::BadImage
image is unbound but the character type for this model is *eNormal*.

ccOCDefs::NotImplemented
image is bound but the character type for this model is *eBlank*.

ccOCDefs::BadParams
image and *mask* are both bound but their window regions are unequal; or the *image* or *mask* is bound but the specified image area does not fit within the window region.

---

imageArea

```
void imageArea(const ccPelRect& area);

const ccPelRect& imageArea() const;
```

- `void imageArea(const ccPelRect& area);`

Sets the area for this character.

Parameters

area The area for the character in image coordinates.

- `const ccPelRect& imageArea() const;`

Returns the area for this character in image coordinates.

encloseRect

```
ccRect encloseRect(
  const cc2Xform& clientFromModel = cc2Xform(),
  const cc2Xform& modelOffset = cc2Xform(),
  const cc2Xform& clientFromImage = cc2Xform(),
```

```
double translationUncertainty = 0,
const ccRadian& rotationUncertainty = ccRadian(0),
double scaleUncertainty = 0) const;
```

Returns the smallest rectangular area that is guaranteed to enclose the character. The enclosing rectangle is aligned with respect to the specified image coordinates.

Parameters

clientFromModel

A transformation that maps the character model's coordinate system to the client coordinate system.

modelOffset

A transformation that maps the origin of the model to the origin of the client coordinate system.

clientFromImage

A transformation that maps the image coordinate system to the client coordinate system.

translationUncertainty

The character's translation uncertainty, which must be greater than or equal to 0.

rotationUncertainty

The character's rotation uncertainty, which must be greater than or equal to 0.

scaleUncertainty

The character's scale uncertainty, which must be greater than or equal to 0. You should not specify a value greater than 0.10 for the scale uncertainty.

Throws

ccOCDefs::BadParams

The specified value for *translationUncertainty*, *rotationUncertainty*, or *scaleUncertainty* is less than 0.

Typedefs

ccOCModelPtrh

```
typedef ccPtrHandle<ccOCModel> ccOCModelPtrh;
```

ccOCModelPtrh_const

```
typedef ccPtrHandle_const<ccOCModel> ccOCModelPtrh_const;
```

ccOCRClassifier

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifier;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRClassifier is a trainable classifier for classifying rectified images. **ccOCRClassifier** can be trained incrementally, even after **ccOCRClassifier** has been run.

Notes

There is no function called **endTrain()** because the tool is designed to be run at any time and to accept additional training instances at any time (at any time does not imply that the tool is multi-thread safe and re-entrant – it is not re-entrant – the tool follows the convention that different threads can safely run asynchronously on different tool objects, but different threads cannot safely run asynchronously on the same tool objects).

Classifier Training Instances

Typically, the OCR Classifier tool's robustness benefits from training on multiple instances of each character class, with each instance representing normal variations to be expected at run time. The OCR Classifier tool uniquely identifies each such training instance by **key.characterCode()**, **key.instance()**, **key.variant()**, and **key.fontID()**.

Collectively, these values form the identifier of the training instance.

You can provide multiple training instances with arbitrary identifiers. The OCR Classifier tool maintains the identifiers you specify as long as there is no conflict, and would automatically resolve conflicts by assigning new unique instance values when multiple training instances shared the same identifier. In other words, if a new training instance's identifier (*key.characterCode*, *key.instance*, *key.variant*, and *key.fontId*) is already present in the classifier's training instances, then the OCR Classifier tool assigns a new value for *key.instance* of its copy of the **ccOCChar** so that the resulting identifier is unique. It never changes any other part of the identifier.

It is legal to train exactly the same training instance multiple times, although this usually does not help with the tool's performance.

The OCR Classifier tool treats all as separate independent instances, each assigned a unique internal instance number.

Classifier Training

The OCR Classifier tool is designed to be incrementally trainable and to be run only after it has been trained. In addition, the tool's behavior depends on *trainParams*, and, as such, all of the training instances should reflect the same *trainParams*.

Towards this end, OCR Classifier tool training involves two steps:

1. Setting *trainParams*
2. Adding the training instances

As a corollary, the OCR Classifier tool should not accept training instances until after the train parameters have been specified.

The classifier is trained in the following way:

1. You specify the training parameters using

   ccOCRClassifier.startTrain() or **ccOCRClassifier.retrain()**

2. You specify training instances using any overload of

   ccOCRClassifier.trainIncremental().

Then, after training one or more instances, the OCR Classifier tool is able to classify rectified images.

Notes

startTrain() always calls **untrain()** as its first step.

retrain() respects (does not discard) the trained instances.

Classifier Makes Deep Copies of Training Instances

The classifier internally makes deep copies of the training instances.

Classifier Training Usage Examples

Some training usage examples are as follows:

1. You want to train a set of characters:

```
classifier.startTrain(trainParams);

classifier.trainIncremental(trainInstancesVec);
```

2. You want to incrementally train an additional character:

```
classifier.trainIncremental(trainInstance);
```

3. You want to retrain with different training parameters:

```
classifier.retrain(trainParams);
```

4. You want to start over, that is, delete all the training instances and train a new set of character instances (note that this code fragment is identical to the code fragment for case (1) because **startTrain()** internally calls **untrain()** which deletes all of the training instances).

```
classifier.startTrain(trainParams);

classifier.trainIncremental(trainInstancesVec);
```

Convenience Functions for Training

The OCR Classifier API also provides 2 convenience **train()** functions that call **startTrain()** followed by an incremental training function.

Discarding Training Instances to Conserve Memory

Nominally, **ccOCRClassifier** keeps a copy of the training instances. For memory conservation purposes, the classifier also has the ability to purge this copy of training instances. The flag **saveTrainCharacters()** controls whether training instances should be saved. The drawback of purging the training instances is that the classifier can no longer be retrained. The OCR Classifier tool throws an exception if you call **retrain()** a classifier for which the training instances have been purged.

Notes

When **untrain()** is called, **saveTrainCharacters()** is preserved.

Constructors/Destructors

ccOCRClassifier

```
ccOCRClassifier();

ccOCRClassifier(const ccOCRClassifier& rhs);
```

- `ccOCRClassifier();`

  Constructs an untrained classifier.

- `ccOCRClassifier(const ccOCRClassifier& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

ccOCRClassifier

```
~ccOCRClassifier();
```

Destructor.

Public Member Functions

isTrained

```
bool isTrained() const;
```

Gets whether this classifier has been trained with at least one training instance.

isStartTrained

```
bool isStartTrained() const;
```

Gets whether training has started, regardless of whether the classifier has received any training instances.

Notes

startTrain(), **train()**, and **retrain()** turn this to true, and **untrain()** turns this to false.

untrain

```
void untrain();
```

Untrains the classifier. Discard information of all training parameters and training instances.

Notes

Calling **untrain()** on an untrained classifier has no effect. A classifier is untrained if **isStartTrained()** == false.

After the call, **isTrained()** and **isStartTrained()** become/remain false.

This does not change the value of **saveTrainCharacters()**.

startTrain

```
void startTrain(const ccOCRClassifierTrainParams& params,
 ccDiagObject* pDiagObj = NULL,
 c_UInt32 diagFlags = 0);
```

Starts the training process with the specified parameters.

Parameters

params	The parameters.
pDiagObj	An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

This calls **untrain()** as the first step.

It is legal to call **startTrain()** repeatedly. Effects of earlier calls are cleared by the last call.

trainIncremental

```
void trainIncremental(const ccOCChar& trainCharacter,
 ccDiagObject* pDiagObj = NULL,
 c_UInt32 diagFlags = 0);

void trainIncremental(
 const cmStd vector<ccOCChar>& trainCharacters,
 ccDiagObject* pDiagObj = NULL,
 c_UInt32 diagFlags = 0);

void trainIncremental(
 const cmStd vector<const ccOCChar*>& trainCharacters,
 ccDiagObject* pDiagObj = NULL,
 c_UInt32 diagFlags = 0);
```

-
```
void trainIncremental(const ccOCChar& trainCharacter,
 ccDiagObject* pDiagObj = NULL,
 c_UInt32 diagFlags = 0);
```

Adds one training instance to this classifier.

Parameters

trainCharacter The additional training instance.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

This requires **isStartTrained()** be true.

After the call, **isTrained()** becomes/remains true.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained() == false.

ccOCRClassifierDefs::BadParams
trainCharacter's *cellRectImage* is not specified, or
trainCharacter.key().characterCode() ==
ccCharCode::eUnknown

ccOCRClassifierDefs::CannotTrain
trainCharacter.key().isSpace() !=
trainCharacter.metrics().isBlank() , or
saveTrainCharacters()==false for algorithms that must train on all training characters at once.

Notes

The algorithms *eBasic* and *eBasicWithValidation* can continue incremental training when **saveTrainCharacters()**==false.

- ```
  void trainIncremental(
    const cmStd vector<ccOCChar>& trainCharacters,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0);
  ```

Adds training instances to this classifier.

Parameters

trainCharacters The additional training instances.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:
  ```

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

This requires **isStartTrained()** be true.

After the call, **isTrained()** becomes/remains true.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained() == false.

ccOCRClassifierDefs::BadParams
Any of the following is true:

- **trainCharacters.size()** returns 0.
- Any element of *trainCharacters* is NULL.
- For any element of *trainCharacters*, the *trainCharacter*'s *cellRectImage* is not specified.
- For any element of *trainCharacters*, **trainCharacter.key().characterCode()** == *ccCharCode::eUnknown*
- For any element of *trainCharacters*, **trainCharacter.key().isSpace()** != **trainCharacter.metrics().isBlank()**

ccOCRClassifierDefs::CannotTrain
saveTrainCharacters()==false for algorithms that must train on all training characters at once.

Notes

The algorithms *eBasic* and *eBasicWithValidation* can continue incremental training when **saveTrainCharacters()**==false.

- ```
  void trainIncremental(
    const cmStd vector<const ccOCChar*>& trainCharacters,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0);
  ```

  Adds training instances to this classifier.
  ```

Parameters

trainCharacters The additional training instances.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

This requires **isStartTrained()** be true.

After the call, **isTrained()** becomes/remains true.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained() == false.

ccOCRClassifierDefs::BadParams
Any of the following is true:

- **trainCharacters.size()** returns 0.
- For any element of *trainCharacters*, the *trainCharacter*'s *cellRectImage* is not specified.
- For any element of *trainCharacters*, **trainCharacter.key().characterCode()** == *ccCharCode::eUnknown*
- For any element of *trainCharacters*, **trainCharacter.key().isSpace()** != **trainCharacter.metrics().isBlank()**

ccOCRClassifierDefs::CannotTrain
saveTrainCharacters()==false for algorithms that must train on all training characters at once.

Notes

The algorithms *eBasic* and *eBasicWithValidation* can continue incremental training when **saveTrainCharacters()**==false.

train

```
void train(const ccOCRClassifierTrainParams& trainParams,
  const cmStd vector<ccOCChar>& trainCharacters,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0);

void train(const ccOCRClassifierTrainParams& trainParams,
  const cmStd vector<const ccOCChar*>& trainCharacters,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0);
```

- ```
  void train(const ccOCRClassifierTrainParams& trainParams,
    const cmStd vector<ccOCChar>& trainCharacters,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0);
  ```

  Convenience function to call **startTrain()** followed by **trainIncremental()**.

  Parameters

  trainParams The training parameters.

  trainCharacters The training characters.

  pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccOCRClassifierDefs::BadParams
  if any of the following is true:

  - **trainCharacters.size()** returns 0.
  - For any element of *trainCharacters*, the *trainCharacter*'s *cellRectImage* is not specified.
  - For any element of *trainCharacters*, **trainCharacter.key().characterCode()** == *ccCharCode::eUnknown*
  ```

- For any element of *trainCharacters*, **trainCharacter.key().isSpace()** != **trainCharacter.metrics().isBlank()**

ccOCRClassifierDefs::CannotTrain
saveTrainCharacters()==false for algorithms that must train on all training characters at once.

- ```
  void train(const ccOCRClassifierTrainParams& trainParams,
    const cmStd vector<const ccOCChar*>& trainCharacters,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0);
  ```

Convenience function to call **startTrain()** followed by **trainIncremental()**.

Parameters

trainParams The training parameters.

trainCharacters The training characters.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccOCRClassifierDefs::BadParams
if any of the following is true:

- **trainCharacters.size()** returns 0.
- Any element of *trainCharacters* is NULL.
- For any element of *trainCharacters*, the *trainCharacter*'s *cellRectImage* is not specified.
- For any element of *trainCharacters*, **trainCharacter.key().characterCode()** == *ccCharCode::eUnknown*
- For any element of *trainCharacters*, **trainCharacter.key().isSpace()** != **trainCharacter.metrics().isBlank()**
  ```

ccOCRClassifierDefs::CannotTrain
saveTrainCharacters()==false for algorithms that must train on all training characters at once.

Notes

After the call, **isStartTrained()** and **isTrained()** become/remain true.

It is OK to call **train()** repeatedly. Effects of earlier calls are cleared by the last call.

The algorithms *eBasic* and *eBasicWithValidation* can continue incremental training when **saveTrainCharacters()**==false.

retrain

```
void retrain(const ccOCRClassifierTrainParams& params,
ccDiagObject* pDiagObj = NULL, c_UInt32 diagFlags = 0);
```

Re-train the classifier with the specified parameters and the training instances saved in the tool.

Parameters

params The parameters.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

It is OK to call **retrain()** repeatedly. Effects of earlier calls will be cleared by the last call.

After the call, both **isStartTrained()** and **isTrained()** remain true.

This requires that **saveTrainCharacters()** be true.

This does not change the value of **saveTrainCharacters()**.

Throws

ccOCRClassifierDefs::NotTrained
isTrained()==false.

ccOCRClassifierDefs::NoSavedData
saveTrainCharacters()==false.

run

```
void run(const ccOCChar& runCharacter,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRClassifierRunParams& runParams,
  ccOCRClassifierPositionResult& result,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0) const;

void run(const cmStd vector<ccOCChar>& runCharacters,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRClassifierRunParams& runParams,
  ccOCRClassifierLineResult& result,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0) const;

void run(
  const cmStd vector<const ccOCChar*>& runCharacters,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRClassifierRunParams& runParams,
  ccOCRClassifierLineResult& result,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0) const;

void run(const ccOCChar& runCharacter,
  bool isSpace,
  double spaceScore,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRClassifierRunParams& runParams,
  ccOCRClassifierPositionResult& result,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0) const;

void run(const cmStd vector<ccOCChar>& runCharacters,
  const cmStd vector<c_Bool>& isSpace,
  const cmStd vector<double>& spaceScore,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRClassifierRunParams& runParams,
  ccOCRClassifierLineResult& result,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0) const;

void run(
  const cmStd vector<const ccOCChar*>& runCharacters,
  const cmStd vector<c_Bool>& isSpace,
  const cmStd vector<double>& spaceScore,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRClassifierRunParams& runParams,
```

```
  ccOCRClassifierLineResult& result,
  ccDiagObject* pDiagObj = NULL,
  c_UInt32 diagFlags = 0) const;
```

- ```
  void run(const ccOCChar& runCharacter,
    const ccOCSwapCharSet& swapCharacterSet,
    const ccOCRClassifierRunParams& runParams,
    ccOCRClassifierPositionResult& result,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0) const;
  ```

  Runs the classifier on the input character.

  Parameters

  runCharacter The run-time character.

  swapCharacterSet
  The swap character set.

  runParams The run-time parameters.

  result The result.

  pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

- ```
  void run(const cmStd vector<ccOCChar>& runCharacters,
    const ccOCSwapCharSet& swapCharacterSet,
    const ccOCRClassifierRunParams& runParams,
    ccOCRClassifierLineResult& result,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0) const;
  ```

  Runs the classifier on the input characters.

  Parameters

  runCharacters The run-time characters.

swapCharacterSet
The swap character set.

runParams The run-time parameters.

result The result.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void run(
    const cmStd vector<const ccOCChar*>& runCharacters,
    const ccOCSwapCharSet& swapCharacterSet,
    const ccOCRClassifierRunParams& runParams,
    ccOCRClassifierLineResult& result,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0) const;
  ```

  Runs the classifier on the input characters.

  Parameters

  runCharacters The run-time characters.

  swapCharacterSet
  The swap character set.

  runParams The run-time parameters.

  result The result.

  pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:
  ```

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void run(const ccOCChar& runCharacter,
    bool isSpace,
    double spaceScore,
    const ccOCSwapCharSet& swapCharacterSet,
    const ccOCRClassifierRunParams& runParams,
    ccOCRClassifierPositionResult& result,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0) const;
  ```

  Runs the classifier on the input character.

  Parameters

  runCharacter The run-time character.

  isSpace Whether this character is a space.

  spaceScore Space score (if this character is a space).

  swapCharacterSet
  The swap character set.

  runParams The run-time parameters.

  result The result.

  pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff
  ```

- ```
  void run(const cmStd vector<ccOCChar>& runCharacters,
    const cmStd vector<c_Bool>& isSpace,
    const cmStd vector<double>& spaceScore,
    const ccOCSwapCharSet& swapCharacterSet,
    const ccOCRClassifierRunParams& runParams,
    ccOCRClassifierLineResult& result,
    ccDiagObject* pDiagObj = NULL,
    c_UInt32 diagFlags = 0) const;
  ```

  Runs the classifier on the input characters.

  Parameters

  runCharacters The run-time characters.

  isSpace Whether this character is a space.

  spaceScore Space score (if this character is a space).

  swapCharacterSet
  The swap character set.

  runParams The run-time parameters.

  result The result.

  pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

- ```
  void run(
    const cmStd vector<const ccOCChar*>& runCharacters,
    const cmStd vector<c_Bool>& isSpace,
    const cmStd vector<double>& spaceScore,
    const ccOCSwapCharSet& swapCharacterSet,
    const ccOCRClassifierRunParams& runParams,
  ```

```
ccOCRClassifierLineResult& result,
ccDiagObject* pDiagObj = NULL,
c_UInt32 diagFlags = 0) const;
```

Runs the classifier on the input characters.

Parameters

runCharacters The run-time characters.

isSpace Whether this character is a space.

spaceScore Space score (if this character is a space).

swapCharacterSet
The swap character set.

runParams The run-time parameters.

result The result.

pDiagObj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

If the input (run-time) character is to be classified as a space, set *isSpace* to true and provide a *spaceScore* within [0, 1].

The classifier ignores *spaceScore* if *isSpace* is false.

The classifier throws an exception if the value of *isSpace* is different from **runCharacter.key().isSpace()**, or is different from **runCharacter.metrics().isBlank()**.

The overloads that do not take the *isSpace* parameter are used only for classifying nonspace characters. For each input character, both **runCharacter.metrics().isBlank()** and **runCharacter.key().isSpace()** must return false.

Throws

ccOCRClassifierDefs::NotTrained
isTrained() == false.

ccOCRClassifierDefs::BadParams
Any pointer is NULL in the overloads that take a vector of pointers.
OR
The vectors do not have the same size in the overloads that take multiple vectors.
OR
The following is true of any *runCharacter*:

- *runCharacter*'s image is unbound.
- *runCharacter*'s *cellRectImage* is not specified.
- **runCharacter.metrics().isBlank()** == true or **runCharacter.key().isSpace()** == true for the overloads that do not take the *isSpace* parameter.
- *isSpace* != **runCharacter.key().isSpace()** or *isSpace* != **runCharacter.metrics().isBlank()** for the overloads that take the *isSpace* parameter.

trainParams

```
const ccOCRClassifierTrainParams& trainParams() const;
```

Gets the training parameters specified when calling **startTrain()**, **train()**, or **retrain()**.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained() == false.

saveTrainCharacters

```
bool saveTrainCharacters() const;

void saveTrainCharacters(bool save);
```

- `bool saveTrainCharacters() const;`

  Gets whether to save all training characters.

- `void saveTrainCharacters(bool save);`

  Sets whether to save all training characters.

Parameters

save Whether to save all training characters.

Notes

When setting this to false, the setter will immediately discard any saved training characters.

After setting this to false, it is not allowed to call the following functions:

- **trainIncremental()** for algorithms needing all characters
- **retrain()**
- **trainCharacters()**
- **trainCharactersProcessed()**

The default value is true.

trainCharacters

```
const cmStd vector<ccOCChar>& trainCharacters() const;
```

Gets the training characters.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained() == false.

ccOCRClassifierDefs::NoSavedData
saveTrainCharacters()==false.

trainCharactersProcessed

```
const cmStd vector<ccOCChar>& trainCharactersProcessed()
  const;
```

Gets the training characters after processing (re-sampling and image pre-processing).

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained()==false.

ccOCRClassifierDefs::NoSavedData
saveTrainCharacters()==false.

trainCharacterKeys

```
const cmStd vector<ccOCCharKey>& trainCharacterKeys()
  const;
```

Gets the keys of all training characters.

Notes

This data is available even if **saveTrainCharacters()** == false.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained()==false.

getTrainCharacterIndices

```
cmStd vector<c_Int32> getTrainCharacterIndices(
  const ccOCCharKey& key) const;
```

Returns the indices of all training instances in **trainCharacterKeys()** that match the specified character key.

Parameters

key The character key.

Notes

This function attempts to match all specified fields of **ccOCCharKey**. If any of *key.instance*, *key.variant*, and *key.fontID* is equal to *ccOCCharKey::eUnspecified*, it is treated as *don't-care*, and this function ignores that field.

Returns an empty vector if the specified key is not found.

Throws

ccOCRClassifierDefs::NotStartTrained
isStartTrained()==false.

ccOCRClassifierDefs::BadParams
key.characterCode() == *ccCharCode::eUnknown*

Operators

operator=

```
ccOCRClassifier& operator=(const ccOCRClassifier& rhs);
```

Assignment operator.

Parameters

rhs The object to assign to this one.

ccOCRClassifierCharResult

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifierCharResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains result data for one potential classification of the run-time image.

Constructors/Destructors

ccOCRClassifierCharResult

```
ccOCRClassifierCharResult();
```

Constructs an object with default values.

isComputed = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Gets whether this object contains valid data.

Notes

All other getters throw *NotComputed* if **isComputed()** == false.

key

```
const ccOCCharKey& key() const;
```

Gets the classification result character key.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

score

```
double score() const;
```

Gets the classification result score.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

isPrimarySwap

```
bool isPrimarySwap() const;
```

Gets whether the character in this result is a swap character of the primary character.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

Operators

operator==

```
bool operator==(const ccOCRClassifierCharResult& rhs)
const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCRClassifierDefs

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifierDefs;
```

This class is a namespace to declare enumerations that are shared by the classes of the OCR Classifier tool.

Enumerations

PositionStatus

```
enum PositionStatus;
```

This enumeration defines the classification status.

Value	Meaning
eRead = 1	Classified with good confidence.
eConfused = 2	Classified with poor confidence.
eFailed = 3	Failed to classify.

ConfusionExplanation

```
enum ConfusionExplanation;
```

This enumeration defines the confusion explanation.

Value	Meaning
eNotConfused = 0	No confusion.
eConfidenceScoreTooLow = 1	Low confidence score.
eClassificationValidationFailure = 2	Failed classification validation.

Algorithm

```
enum Algorithm;
```

This enumeration defines the classification algorithm to be used.

Value	Meaning
eBasic = 0x1	The basic classification algorithm.

Value	Meaning
eBasicWithValidation = 0x2	The basic classification algorithm with the classification validation step.
kDefaultAlgorithm = *eBasic*	The default classification algorithm.

ImagePreprocessing

```
enum ImagePreprocessing;
```

This enumeration defines the input image's preprocessing method.

Value	Meaning
eNormalizeHistogram = 0x1	Normalize the input image's histogram.
eSubtractMedian = 0x2	Subtract the median of a local neighborhood.
kDefaultImagePreprocessing = *eNormalizeHistogram*	The default preprocessing method.

Notes

If both histogram normalization and local neighborhood median subtraction are enabled, histogram normalization happens after median subtraction.

ccOCRClassifierLineResult

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifierLineResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains all result data for the classification of one line of characters.

Constructors/Destructors

ccOCRClassifierLineResult

```
ccOCRClassifierLineResult();
```

Constructs an object with default values.

isComputed = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Gets whether this object contains valid data.

Notes

All other getters throw **NotComputed if isComputed()**==false.

status

```
ccOCRClassifierDefs::PositionStatus status() const;
```

Gets the result status: *eRead*, *eConfused*, or *eFailed*.

Notes

This is the lowest grade among all position results in this line result.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

positionResults

```
const cmStd vector<ccOCRClassifierPositionResult>&
 positionResults() const;
```

Gets all position results in this line result.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

constrainCharacterCodes

```
ccOCRClassifierLineResult constrainCharacterCodes(
 const cmStd vector<cmStd vector<c_Char32> >
  &allowableCharacterCodes,
 const ccOCSwapCharSet &swapCharSet = ccOCSwapCharSet(),
 bool computeConfusionUsingOnlyAllowableCharacters = true)
 const;
```

Computes a new result based on the old result and a specified 2D set of allowable character codes. You also specify whether the confusion should be computed using only the allowable characters or using all the trained characters for the classifier.

If an element of the input *allowableCharacterCodes vector* is empty, then that particular input result is excluded from the output.

Parameters

allowableCharacterCodes
The allowable character codes.

swapCharSet The swap character set.

computeConfusionUsingOnlyAllowableCharacters
If true, confusion is computed using only the allowable characters.

Notes

If the confusion is not limited to only allowable characters, that is, *computeConfusionUsingOnlyAllowableCharacters* == false, then all the non-primary characters will be included as alternative characters (regardless of whether they agree with the fielding).
In other words, if *computeConfusionUsingOnlyAllowableCharacters* == false, then only the primary character is constrained to agree with the fielding.

This function respects the following reserved character codes:

- *ccCharCode::eAnyCharacter*
- *ccCharCode::eAnyNonspaceCharacter*

If the allowable characters include *ccCharCode::eAnyCharacter*, then the associated input *positionResult* is simply copied to the associated computed *positionResult* result.

For each position, if the highest scoring character that satisfies the *allowableCharacterCodes* constraints does not satisfy the accept threshold, then the status is set to *eFailed*. Correspondingly, if the status is *eFailed*, then the primary character is unknown.

If *computeConfusionUsingOnlyAllowableCharacters* == false, then for each position, the confusion character is set to the highest scoring non-primary character; consequently, for each position, *confusionCharacter* may be valid even if the primary character is unknown and the classifier status is failed.

For each position, if the primary character is the only matching character, the confusion character is set to a default constructed character.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

ccOCRDefs::BadParams
allowableCharacterCodes.size() != **positionResults().size()** or *allowableCharacterCodes* includes any Cognex reserved character codes other than *ccCharCode::eAnyCharacter* or *ccCharCode::eAnyNonspaceCharacter*.

Operators

operator==

```
bool operator==(const ccOCRClassifierLineResult& rhs)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCRClassifierLineResult

ccOCRClassifierPositionResult

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifierPositionResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains all result data for the classification of a run-time character.

Constructors/Destructors

ccOCRClassifierPositionResult

```
ccOCRClassifierPositionResult();
```

Constructs an object with default values.

isComputed = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Gets whether this object contains valid data.

Notes

All other getters throw **NotComputed if isComputed()**==false.

primaryCharacter

```
const ccOCRClassifierCharResult& primaryCharacter() const;
```

Gets the determined classification of the input image.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

status

```
ccOCRClassifierDefs::PositionStatus status() const;
```

Gets the result status: *eRead*, *eConfused*, or *eFailed*.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

confusionExplanation

```
ccOCRClassifierDefs::ConfusionExplanation
 confusionExplanation() const;
```

Gets a reason that leads to the *eConfused* status.

Notes

This is useful only if **status()** returns *eConfused*.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

confusionCharacter

```
const ccOCRClassifierCharResult& confusionCharacter()
 const;
```

Gets the confusion character.

Notes

This is set to a default **ccOCRClassifierCharResult** if there is no confusion character (that is, when **alternativeCharacters()** returns an empty vector).

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

confidenceScore

```
double confidenceScore() const;
```

Gets the confidence score, in the range [0, 1].

Notes

The confidence score is the difference between **primaryCharacter().score()** and **confusionCharacter().score()**.

This is set to 0 if **confusionExplanation()** returns *eClassificationValidationFailure*.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

alternativeCharacters

```
const cmStd vector<ccOCRClassifierCharResult>&
  alternativeCharacters() const;
```

Gets the list of alternative classes, and optionally, their training instances, which induced scores satisfying the alternative score threshold.

Notes

This list includes at most one instance for each character code/class.

This list is sorted in the order of descending scores.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

skippedTrainCharacterIndices

```
const cmStd vector<c_Int32>&
skippedTrainCharacterIndices() const;
```

Gets the list of indices to training character instances that were skipped (that is, not considered) during classification, for example, due to violating scale/size constraints.

Notes

This vector would be empty if **runParams.reportSkippedTrainCharacterIndices()** was false for the run that generated this result.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

processedImage

```
ccPelBuffer_const<c_UInt8> processedImage() const;
```

Gets the processed image, which is generated from the rectified input image and used for classification.

Notes

This image would be unbound if **runParams.keepProcessedImage()** was false for the run that generated this result.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

constrainCharacterCodes

```
ccOCRClassifierPositionResult constrainCharacterCodes(
  const cmStd vector<c_Char32> &allowableCharacterCodes,
  const ccOCSwapCharSet &swapCharSet = ccOCSwapCharSet(),
  bool computeConfusionUsingOnlyAllowableCharacters = true)
  const;
```

Computes a new result based on the old result and a specified set of allowable character codes. You also specify whether the confusion should be computed using only the allowable characters or using all the trained characters for the classifier.

Parameters

allowableCharacterCodes
The allowable character codes.

swapCharSet The swap character set.

computeConfusionUsingOnlyAllowableCharacters
If true, confusion is computed using only the allowable characters.

Notes

This function respects the following reserved character codes:

- *ccCharCode::eAnyCharacter*
- *ccCharCode::eAnyNonspaceCharacter*

If the allowable characters include *ccCharCode::eAnyCharacter*, then the input result is simply copied to the output result.

If the confusion is not limited to only allowable characters, that is, *computeConfusionUsingOnlyAllowableCharacters* == false, then all the non-primary characters will be included as alternative characters (regardless of whether they agree with the fielding).
In other words, if *computeConfusionUsingOnlyAllowableCharacters* == false, then only the primary character is constrained to agree with the fielding.

If the highest scoring character that satisfies the *allowableCharacterCodes* constraints does not satisfy the accept threshold, then the status is set to *eFailed*. Correspondingly, if the status is *eFailed*, then the primary character is unknown.

If *computeConfusionUsingOnlyAllowableCharacters* == false, then the confusion character is set to the highest scoring non-primary character; consequently, *confusionCharacter* may be valid even if the primary character is unknown and the classifier status is failed.

If the primary character is the only matching character, the confusion character is set to a default constructed character.

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

ccOCRDefs::BadParams
allowableCharacterCodes.size() == 0
or *allowableCharacterCodes* includes any Cognex reserved character codes other than *ccCharCode::eAnyCharacter* or *ccCharCode::eAnyNonspaceCharacter*.

acceptThreshold

```
double acceptThreshold() const;
```

Returns the accept threshold used to generate this result

Throws

ccOCRClassifierDefs::NotComputed
isComputed() == false.

Operators

operator==

```
bool operator==(const ccOCRClassifierPositionResult& rhs)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCRClassifierPositionResult

ccOCRClassifierRunParams

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifierRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the run-time parameters for the OCR Classifier tool.

Constructors/Destructors

ccOCRClassifierRunParams

```
ccOCRClassifierRunParams();
```

Constructs an object with default values, which are the following:

- *acceptThreshold* = 0.5
- *confidenceThreshold* = 0.0
- *xScaleFilter* = **ccRange(***0.666, 1.5***)**
- *yScaleFilter* = **ccRange(***0.666, 1.5***)**
- *reportSkippedTrainCharacterIndices* = false
- *keepProcessedImage* = false

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

acceptThreshold

```
double acceptThreshold() const;
void acceptThreshold(double thresh);
```

- `double acceptThreshold() const;`

  Gets the accept threshold.

- `void acceptThreshold(double thresh);`

  Sets the accept threshold.

  Parameters
  thresh The accept threshold.

  Throws
  ccOCRClassifierDefs::BadParams
  thresh < 0 or *thresh* > 1.0

  The default value is 0.5.

confidenceThreshold

```
double confidenceThreshold() const;
void confidenceThreshold(double thresh);
```

- `double confidenceThreshold() const;`

  Gets the confidence threshold.

- `void confidenceThreshold(double thresh);`

  Sets the confidence threshold.

  Parameters
  thresh The confidence threshold.

  Throws

ccOCRClassifierDefs::BadParams
thresh < 0 or *thresh* > 1.0

The default value is 0.0.

useXScaleFilter

```
bool useXScaleFilter() const;
void useXScaleFilter(bool use);
```

- `bool useXScaleFilter() const;`

  Gets whether to use the x-scale filter.

- `void useXScaleFilter(bool use);`

  Sets whether to use the x-scale filter.

  Parameters
  use Whether to use the x-scale filter.

  The default value is true.

xScaleFilter

```
const ccRange& xScaleFilter() const;
void xScaleFilter(const ccRange& range);
```

- `const ccRange& xScaleFilter() const;`

  Gets the x-scale filter range for skipping candidate classes/instances whose rectified training image's x-size (that is, width) is beyond the range specified here.

- `void xScaleFilter(const ccRange& range);`

  Sets the x-scale filter range for skipping candidate classes/instances whose rectified training image's x-size (that is, width) is beyond the range specified here.

  Parameters
  range The x-scale filter range.

  Throws
  ccOCRClassifierDefs::BadParams
  range is an empty range or a full range, or
  range.start() < 0

Notes

X-scale is computed as the x-size of rectified run-time image divided by the x-size of a particular training class/instance.

This is effective only if **useXScaleFilter()** returns true.

The default range is (0.666, 1.5).

useYScaleFilter

```
bool useYScaleFilter() const;
void useYScaleFilter(bool use);
```

- `bool useYScaleFilter() const;`

  Gets whether to use the y-scale filter.

- `void useYScaleFilter(bool use);`

  Sets whether to use the y-scale filter.

  Parameters

  use Whether to use the y-scale filter.

  The default value is true.

yScaleFilter

```
const ccRange& yScaleFilter() const;
void yScaleFilter(const ccRange& range);
```

- `const ccRange& yScaleFilter() const;`

  Gets the y-scale filter range for skipping candidate classes/instances whose rectified training image's y-size (that is, height) is beyond the range specified here.

- `void yScaleFilter(const ccRange& range);`

  Sets the y-scale filter range for skipping candidate classes/instances whose rectified training image's y-size (that is, height) is beyond the range specified here.

  Parameters

  range The y-scale filter range.

Throws

ccOCRClassifierDefs::BadParams

range is an empty range or a full range, or
range.start() < 0

Notes

Y-scale is computed as the y-size of rectified run-time image divided by the y-size of a particular training class/instance.

This is effective only if **useYScaleFilter()** returns true.

The default range is (0.666, 1.5).

reportSkippedTrainCharacterIndices

```
bool reportSkippedTrainCharacterIndices() const;

void reportSkippedTrainCharacterIndices(bool report);
```

- `bool reportSkippedTrainCharacterIndices() const;`

  Gets whether to include in the result object the set of train character indices that were skipped because they did not satisfy the scale constraints.

- `void reportSkippedTrainCharacterIndices(bool report);`

  Sets whether to include in the result object the set of train character indices that were skipped because they did not satisfy the scale constraints.

  Parameters

report	Whether to include in the result object the set of train character indices that were skipped because they did not satisfy the scale constraints.

  The default value is false.

keepProcessedImage

```
bool keepProcessedImage() const;

void keepProcessedImage(bool keep);
```

- `bool keepProcessedImage() const;`

  Gets whether to keep around the processed image corresponding to the run-time rectified input image.

- `void keepProcessedImage(bool keep);`

  Sets whether to keep around the processed image corresponding to the run-time rectified input image.

  This processed image is usually provided for diagnostic purposes and is usually disabled for production runs.

  Parameters

keep	Whether to keep around the processed image corresponding to the run-time rectified input image.

  The default value is false.

Operators

operator== `bool operator==(const ccOCRClassifierRunParams& rhs) const;`

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCRClassifierTrainParams

```
#include <ch_cvl/ocrclass.h>

class ccOCRClassifierTrainParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains training parameters for the OCR Classifier tool.

Constructors/Destructors

ccOCRClassifierTrainParams

```
ccOCRClassifierTrainParams();
```

Constructs an object with default values, which are the following:

- *templateSize* = (10, 18)
- *maintainAspectRatio* = false
- *algorithm* = *kDefaultAlgorithm*
- *imagePreprocessing* = *kImageDefaultPreprocessing*

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

templateSize

```
ccIPair templateSize() const;

void templateSize(const ccIPair& size);
```

- ```
  ccIPair templateSize() const;
  ```

  Gets the size to which the rectified images are resampled.
  ```

- `void templateSize(const ccIPair& size);`

  Sets the size to which the rectified images are resampled.

  Parameters

  size The size.

  Throws

  ccOCRClassifierDefs::BadParams
  size.x <= 0 or *size.y* <= 0

  ccOCDefs::NotImplemented
  size.x * *size.y* > 32768

  The default size is (10, 18).

maintainAspectRatio

```
bool maintainAspectRatio() const;
void maintainAspectRatio(bool maintain);
```

- `bool maintainAspectRatio() const;`

  Gets whether to maintain the rectified images' aspect ratio when resampling the rectified image at train-time/run-time.

- `void maintainAspectRatio(bool maintain);`

  Sets whether to maintain the rectified images' aspect ratio when resampling the rectified image at train-time/run-time.

  Parameters

  maintain Whether to maintain the rectified images' aspect ratio when resampling the rectified image at train-time/run-time.

  The default value is false.

algorithm

```
ccOCRClassifierDefs::Algorithm algorithm() const;
void algorithm(ccOCRClassifierDefs::Algorithm algorithm);
```

- `ccOCRClassifierDefs::Algorithm algorithm() const;`

  Gets the algorithm used for classification.

- `void algorithm(ccOCRClassifierDefs::Algorithm algorithm);`

  Sets the algorithm used for classification.

  Parameters
  algorithm The classification algorithm.

  Throws
  ccOCRClassifierDefs::BadParams
  algorithm is not a valid value for *ccOCRClassifierDefs::Algorithm*.

  The default value is *ccOCRClassifierDefs::kDefaultAlgorithm*.

imagePreprocessing

```
c_UInt32 imagePreprocessing() const;
void imagePreprocessing(c_UInt32 preproc);
```

- `c_UInt32 imagePreprocessing() const;`

  Gets the image preprocessing methods used for classification.

- `void imagePreprocessing(c_UInt32 preproc);`

  Sets the image preprocessing methods used for classification.

  The value is a bit field from *ccOCRClassifierDefs::ImagePreprocessing*.

  Parameters
  preproc The image preprocessing methods used for classification.

  Throws
  ccOCRClassifierDefs::BadParams
  preproc is not a bitwise OR of values from *ccOCRClassifierDefs::ImagePreprocessing*.

  Notes
  The same image preprocessing methods are applied to both train-time and run-time characters.

  The result image's client coordinate transform retains any effect that the specified image preprocessing methods may have on the client transform.

  The default value is *ccOCRClassifierDefs::kDefaultImagePreprocessing*.

Operators

operator==

```
bool operator==(const ccOCRClassifierTrainParams& rhs)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCRDefs

```
#include <ch_cvl/ocrdefs.h>

class ccOCRDefs;
```

The **ccOCRDefs** class contains all enumerations specific to the OCR reader supporting classes.

Enumerations

FontCharWidthType

```
enum FontCharWidthType;
```

This enumeration defines the width type of the characters in the font.

Value	Meaning
eFontCharWidthTypeUnknown = 0x1	Unknown width type; the font may be either fixed width or proportional width.
eFontCharWidthTypeFixed = 0x2	All character mark rectangles in the font have the same width.
eFontCharWidthTypeVariable = 0x4	The characters in the font may have mark rectangles with different widths.
kDefaultFontCharWidthType = *eFontCharWidthTypeUnknown*	The default width type.

FontPitchType

```
enum FontPitchType;
```

This enumeration defines the pitch type for the characters in the font.

Value	Meaning
eFontPitchTypeUnknown = 0x1	Unknown pitch type, but the type is expected to be either fixed or proportional, not variable.
eFontPitchTypeFixed = 0x2	The pitch is fixed, which means that the pitch between any pair of characters is constant, for example, regardless of the width of the mark rectangles of the characters. Thus, all characters can be considered to have cell rectangles that are the same width. How the pitch is measured for a given fixed-pitch font is specified by the pitch metric.
eFontPitchTypeProportional = 0x4	The pitch is proportional, which means that the pitch between any pair of characters depends on the particular characters. The width of a character's cell rectangle is typically approximately proportional to the width of its mark rectangle. **Note**: Although no pitch measurement is constant throughout a string, typically the intercharacter gap, which is the distance from the right side of one character's mark rectangle to the left side of the adjacent character's mark rectangle, is approximately constant.

Value	Meaning
eFontPitchTypeVariable = 0x8	No character-to-character distance metric is consistent throughout a string; character placement is erratic. The pitch is neither fixed nor proportional. **Note**: Variable pitch is different from unknown pitch, because unknown pitch means that the pitch is either fixed or proportional, but it is not known which one.
kDefaultFontPitchType = *eFontPitchTypeUnknown*	The default pitch type.

FontPitchMetric

```
enum FontPitchMetric;
```

This enumeration defines the pitch metric for characters in the font.

Value	Meaning
eFontPitchMetricUnknown = 0x1	Unknown metric; the appropriate pitch may be any of the other pitch metrics, or else there is no consistent pitch metric (as may be the case for a proportional-pitch font).
eFontPitchMetricLeftToLeft = 0x2	Measure pitch as the distance from the left side of a character's mark rectangle to the left side of the adjacent character's mark rectangle. Thus, the character's mark rectangle appears left-justified within the character's cell rectangle.
eFontPitchMetricCenterToCenter = 0x4	Measure pitch as the distance from the center of a character's mark rectangle to the center of the adjacent character's mark rectangle. Thus, the character's mark rectangle appears centered within the character's cell rectangle.

Value	Meaning
eFontPitchMetricRightToRight = 0x8	Measure pitch as the distance from the right side of a character's mark rectangle to the right side of the adjacent character's mark rectangle. Thus, the character's mark rectangle appears right-justified within the character's cell rectangle.
kDefaultFontPitchMetric = *eFontPitchMetricUnknown*	The default pitch metric.

ccOCRDictionaryChar

```
#include <ch_cvl/ocrdcbas.h>

class ccOCRDictionaryChar;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRDictionaryChar represents a single character used by the OCR Dictionary Fielding tool.

Constructors/Destructors

cOCRDictionaryChar

```
explicit ccOCRDictionaryChar(
  c_Char32 characterCode = ccCharCode::eUnknown,
  double score = 0.0);
```

Constructs a character.

Parameters

characterCode The character code to construct.

score The character score, which must be in the range 0.0 through 1.0;

Throws

ccOCRDictionaryDefs::BadParams
score is less than 0.0 or greater than 1.0.

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

characterCode

```
c_Char32 characterCode() const;
void characterCode(c_Char32 characterCode);
```

- `c_Char32 characterCode() const;`

  Returns this object's character code.

- `void characterCode(c_Char32 characterCode);`

  Sets this object's character code. The character code is generally a Unicode code point, although the **ccCharCode** class defines several reserved values.

  The default value for **characterCode** is *ccCharCode::eUnknown*.

  Parameters
  characterCode The character code.

score

```
double score() const;
void score(double score);
```

- `double score() const;`

  Returns this character's score, a value in the range 0.0 through 1.0.

- `void score(double score);`

  Sets this character's score. The default value for **score** is 0.0.

  Parameters
  score The score to set.

  Throws
  ccOCRDictionaryDefs::BadParams
  score is less than 0.0 or greater than 1.0.

Operators

operator char() const;

```
operator char() const;
```

Return the **char** representation of this object.

Throws

ccCharCodeDefs::NotRepresentable
The character code cannot be represented as a **char**.

operator wchar_t() const;

```
operator wchar_t() const;
```

Return the **wchar_t** representation of this object.

Throws

ccCharCodeDefs::NotRepresentable
The character code cannot be represented as a **wchar_t**.

operator==

```
bool operator==(const ccOCRDictionaryChar& rhs) const;
```

Returns true if this **ccOCRDictionaryChar** is equal to the supplied one, false otherwise.

Parameters

rhs The **ccOCRDictionaryChar** to compare to this one.

operator!=

```
bool operator!=(const ccOCRDictionaryChar& rhs) const;
```

Returns false if this **ccOCRDictionaryChar** is equal to the supplied one, true otherwise.

Parameters

rhs The **ccOCRDictionaryChar** to compare to this one.

ccOCRDictionaryChar

ccOCRDictionaryCharMulti

```
#include <ch_cvl/ocrdcbas.h>

class ccOCRDictionaryCharMulti;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRDictionaryCharMulti represents a set of characters used by the OCR Dictionary Fielding tool. This class is usually used to represent the results of the OCR Classifier tool for a single character position.

Each **ccOCRDictionaryCharMulti** includes a primary character along with 0 or more alternative characters. In the case of an OCR Classifier tool result of *eRead* or *eConfused*, the non-primary characters are the alternative characters identified by the classifier.

Constructors/Destructors

ccOCRDictionaryCharMulti

```
explicit ccOCRDictionaryCharMulti(
  const cmStd vector<ccOCRDictionaryChar>& characters=
  cmStd vector<ccOCRDictionaryChar>());

explicit ccOCRDictionaryCharMulti(
  const ccOCRDictionaryChar& c);
```

- ```
  explicit ccOCRDictionaryCharMulti(
    const cmStd vector<ccOCRDictionaryChar>& characters=
    cmStd vector<ccOCRDictionaryChar>());
  ```

  Constructs a **ccOCRDictionaryCharMulti** object from the supplied vector of **ccOCRDictionaryChar**.

  Parameters

  characters The **ccOCRDictionaryChar**s.
  ```

- ```
  explicit ccOCRDictionaryCharMulti(
    const ccOCRDictionaryChar& c);
  ```

  Constructs a **ccOCRDictionaryCharMulti** object from the supplied **ccOCRDictionaryChar**.

  Parameters

  c The character.

  Notes

  The compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

characters

```
const cmStd vector<ccOCRDictionaryChar>& characters()
   const;

void characters(
   const cmStd vector<ccOCRDictionaryChar>& c);
```

- ```
  const cmStd vector<ccOCRDictionaryChar>& characters()
     const;
  ```

  Return the individual **ccOCRDictionaryChar** objects in this object.

- ```
  void characters(
     const cmStd vector<ccOCRDictionaryChar>& c);
  ```

  Set the **ccOCRDictionaryChar** objects in this object to the supplied vector.

  Parameters

  c The **ccOCRDictionaryChar**s to set.

isEmpty

```
bool isEmpty() const;
```

Returns *true* if this object contains no individual characters, *false* otherwise.

isSingular

```
bool isSingular() const ;
```

Returns *true* if this object contains a single individual character, *false* if it contains 0 or more than 1 characters.
```

size

```
c_Int32 size() const;
```

Returns the number of individual characters in this object.

primaryCharacter

```
const ccOCRDictionaryChar& primaryCharacter() const;
```

Returns the primary character in the multicharacter class. The primary character is the first character in **ccOCRDictionaryCharMulti::characters()**.

Throws

ccOCRDictionaryDefs::NotExpressible
This **ccOCRDictionaryCharMulti** contains no characters.

Operators

operator==

```
bool operator==(const ccOCRDictionaryCharMulti& rhs) const;
```

Returns true if this **ccOCRDictionaryCharMulti** is equal to the supplied one, false otherwise.

Parameters

rhs The **ccOCRDictionaryCharMulti** to compare to this one.

operator!=

```
bool operator!=(const ccOCRDictionaryCharMulti& rhs) const;
```

Returns false if this **ccOCRDictionaryCharMulti** is equal to the supplied one, true otherwise.

Parameters

rhs The **ccOCRDictionaryCharMulti** to compare to this one.

ccOCRDictionaryCharMulti

ccOCRDictionaryDefs

```
#include <ch_cvl/ocrddefs.h>

class ccOCRDictionaryDefs;
```

This class is a namespace to declare constants and enumerations.

Enumerations

Fielding

```
enum Fielding;
```

This enumeration provides some pre-defined sets of characters for convenience in specifying the acceptable choices for a character position.

Value	Meaning
eAlphaUppercase = 0x1	Characters A through Z.
eAlphaLowercase = 0x2	Characters a through z.
eNumeric = 0x4	Characters 0 through 9.
eSpace = 0x8	The space character (UTF-32 character code 0x20).
eAnyNonspaceCharacter = 0x10	Any character not in the Cognex-reserved UTF-32 code ranges except the space character whose UTF-32 character code is 0x20. **Note**: This does include other white spaces, for example, character codes 0x09 through 0x0D.
eAnyCharacter = *eSpace* | *eAnyNonspaceCharacter*	Any character not in the Cognex-reserved UTF-32 code ranges.
eAlpha = *eAlphaUppercase* | *eAlphaLowercase*	Characters A through Z and a through z.
eAlphaNumeric = *eAlpha* | *eNumeric*	Characters A through Z, a through z, and 0 through 9.
eAlphaUppercaseNumeric = *eAlphaUppercase* | *eNumeric*	Characters A through Z and 0 through 9.

Value	Meaning
eAlphaLowercaseNumeric = *eAlphaLowercase* | *eNumeric*	Characters a through z and 0-9.
kDefaultFielding = *eAnyNonspaceCharacter*	Default fielding is any non-Cognex-reserved nonspace character.

PositionStatus

```
enum PositionStatus;
```

This enumeration defines quality ratings for characters and strings.

Value	Meaning
ePrimary = 1	Matching the primary character.
ePrimarySwap = 2	Matching a swap character of a primary character.
eSecondary = 3	Matching a secondary character.
eSecondarySwap = 4	Matching a swap character of a secondary character.
eSubstituted = 5	Replaced by a non-input character.
eSubstitutedSpace = 6	Replaced by a space.
eNoMatch = 7	No match allowed by the OCR Dictionary Fielding tool.
eInserted = 8	A character inserted by the OCR Dictionary Fielding tool.
eDeleted = 9	An input character deleted by the OCR Dictionary Fielding tool.
eDeletedIgnored = 10	An input character deleted by the OCR Dictionary Fielding tool as part of the portions to be ignored (for example, prefix).

IsGoodMetric

```
enum IsGoodMetric;
```

This enumeration defines which matches to treat as “good.”

Value	Meaning
ePrimaryMatchesAreGood = 0x3	Matching the primary character or a swap character of the primary character is acceptable.
ePrimaryAndSecondaryMatchesAreGood = 0xF	Matching the primary character or the secondary character or a swap character of the primary character or the secondary character is acceptable.
kDefaultIsGoodMetric = *ePrimaryMatchesAreGood*	The default is *ePrimaryMatchesAreGood*.

ccOCRDictionaryDefs

ccOCRDictionaryFielding

```
#include <ch_cvl/ocrfldng.h>

class ccOCRDictionaryFielding;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class specifies the set of acceptable characters at each character position in a string.

Constructors/Destructors

ccOCRDictionaryFielding

```
explicit ccOCRDictionaryFielding(
  const cmStd vector<ccOCRDictionaryPositionFielding>&
  positionFieldings =
  cmStd vector<ccOCRDictionaryPositionFielding>());
```

Constructs a fielding object from the given vector.

Parameters

positionFieldings Position fieldings vector.

Throws

ccOCRDictionaryDefs::InvalidFielding
isEmpty() is true for any element of *positionFieldings*.

ccOCRDictionaryFielding

```
ccOCRDictionaryFielding(
  const ccOCRDictionaryFielding& rhs);
```

Copy constructor.

Parameters

rhs The source of the copy.

ccOCRDictionaryFielding

```
~ccOCRDictionaryFielding();
```

Destructor.

Public Member Functions

positionFieldings

```
const cmStd vector<ccOCRDictionaryPositionFielding>&
  positionFieldings() const;

void positionFieldings(
  const cmStd vector<ccOCRDictionaryPositionFielding>&
  positionFieldings);
```

- ```
  const cmStd vector<ccOCRDictionaryPositionFielding>&
    positionFieldings() const;
  ```

  Gets the fielding for all character positions.

- ```
  void positionFieldings(
    const cmStd vector<ccOCRDictionaryPositionFielding>&
    positionFieldings);
  ```

  Sets the fielding for all character positions.

  Parameters

  positionFieldings Character positions.

  Throws

  ccOCRDictionaryDefs::InvalidFielding

  isEmpty() is true for any of the position fieldings.

size

```
c_Int32 size() const;
```

Gets the size of the position fielding vector.

positionFielding

```
const ccOCRDictionaryPositionFielding&
  positionFielding(c_Int32 index) const;

void positionFielding(c_Int32 index,
  const ccOCRDictionaryPositionFielding& fielding);
```

- ```
  const ccOCRDictionaryPositionFielding&
    positionFielding(c_Int32 index) const;
  ```

  Gets an individual character fielding.
  ```

Parameters

index The position of the character for which you are getting fielding.

Throws

ccOCRDictionaryDefs::BadParams
index < 0 or *index* >= **size()**

- ```
  void positionFielding(c_Int32 index,
    const ccOCRDictionaryPositionFielding& fielding);
  ```

Sets an individual character fielding.

Parameters

index The position of the character for which you are setting fielding.

fielding The fielding value.

Throws

ccOCRDictionaryDefs::BadParams
index < 0 or *index* >= **size()**

ccOCRDictionaryDefs::InvalidFielding
fielding.isEmpty() returned true.

run

```
void run(
  const ccOCRDictionaryStringMulti& inputStringMulti,
  const ccOCSwapCharSet& swapCharacterSet,
  const ccOCRDictionaryFieldingRunParams& runParams,
  ccOCRDictionaryResultSet& resultSet) const;
```

Checks the input string against the fielding requirements.

Parameters

inputStringMulti The input string.

swapCharacterSet
The swap character set.

runParams The run-time parameters.

resultSet The result set.

Throws

ccOCRDictionaryDefs::BadParams
inputString.size() == 0,
or
runParams.fixedLengthFielding() == false &&
runParams.minFieldLastIndex() >= **size()**,
```

or
runParams.fixedLengthFielding() == false &&
runParams.minStringLength() > **size()**,
or
any of the following is true for any element of *inputStringMulti*:

- Contains no character.
- Contains duplicate characters.
- Contains *ccCharCode::eUnknown* when the number of characters is greater than one.

Requires that all characters in *inputStringMulti* be non-Cognex-reserved, except *ccCharCode::eUnknown*, which is allowed.

Notes

If the fielding is empty, then all characters are accepted. This means that the primary character of an *inputStringMulti* character input (except for *ccCharCode::eUnknown*, which is a special case and induces unmatched) is matched as a primary match, and all the remaining characters in an *inputStringMultiCharacter* are matched as secondary matches.

Operators

operator==

```
bool operator==(const ccOCRDictionaryFielding& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCRDictionaryFielding& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for inequality.

operator=

```
ccOCRDictionaryFielding& operator=(
  const ccOCRDictionaryFielding& rhs);
```

Assignment operator.

Parameters

rhs	The object to assign to this one.

ccOCRDictionaryFielding

ccOCRDictionaryFieldingRunParams

```
#include <ch_cvl/ocrfldng.h>

class ccOCRDictionaryFieldingRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class provides run-time parameters for OCR Dictionary Fielding tool.

Constructors/Destructors

ccOCRDictionaryFieldingRunParams

```
ccOCRDictionaryFieldingRunParams();
```

Constructs an object with default values, which are the following:

- *ignoreUnfieldedSpaces*: true
- *fixedLengthFielding*: true
- *minStringLength*: 1
- *maxStringLength*: *ckMaxInt32*
- *maxFieldFirstIndex*: *ckMaxInt32*
- *minFieldLastIndex*: 0
- *ignoreFailingPrefixSuffix*: true
- *isGoodMetric*: *ccOCRDictionaryDefs::kDefaultIsGoodMetric*

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

ignoreUnfieldedSpaces

```
bool ignoreUnfieldedSpaces() const;
void ignoreUnfieldedSpaces(bool ignore);
```

- `bool ignoreUnfieldedSpaces() const;`

  Gets whether to ignore/skip spaces in the input string where the corresponding fielding settings for those character positions do not allow spaces.

- `void ignoreUnfieldedSpaces(bool ignore);`

  Sets whether to ignore/skip spaces in the input string where the corresponding fielding settings for those character positions do not allow spaces.

  Parameters

ignore	Whether to ignore/skip spaces in the input string where the corresponding fielding settings for those character positions do not allow spaces.

fixedLengthFielding

```
bool fixedLengthFielding() const;
void fixedLengthFielding(bool fixed);
```

- `bool fixedLengthFielding() const;`

  Gets whether to run the OCR Dictionary Fielding tool in the fixed-length mode or the variable-length mode.

- `void fixedLengthFielding(bool fixed);`

  Sets whether to run the OCR Dictionary Fielding tool in the fixed-length mode (or the variable-length mode).

  Parameters

fixed	Whether to run the OCR Dictionary Fielding tool in the fixed-length mode.

Notes

If set to true, the tool runs in the fixed-length mode. In this mode, the tool uses the specified fielding vector in its entirety, and ignores the parameters *minStringLength*, *maxStringLength*, *maxFieldFirstIndex*, *minFieldLastIndex*, and *ignoreFailingPrefixSuffix*.

If set to false, the tool runs in the variable-length mode. In this mode, the tool respects the following parameters in choosing a subsequence of the fielding vector that best fits the input string: *minStringLength*, *maxStringLength*, *maxFieldFirstIndex*, *minFieldLastIndex*, and *ignoreFailingPrefixSuffix*.

minStringLength

```
c_Int32 minStringLength() const;

void minStringLength(c_Int32 minStringLength);
```

- `c_Int32 minStringLength() const;`

  Gets the minimum acceptable string length, inclusive.

- `void minStringLength(c_Int32 minStringLength);`

  Sets the minimum acceptable string length, inclusive.

  Parameters

  minStringLength The minimum acceptable string length.

  Throws

  ccOCRDictionaryDefs::BadParams
  minStringLength < 1

  Notes

  This parameter has no effect if **fixedLengthFielding()** returns true.

  The **run()** function throws *ccOCRDictionaryDefs::BadParams* if *minStringLength* > *maxStringLength*, and **fixedLengthFielding()** returns false.

  The **run()** function throws *ccOCRDictionaryDefs::BadParams* if **fixedLengthFielding()** == false and **minStringLength()** > **fielding.size()**

maxStringLength

```
c_Int32 maxStringLength() const;

void maxStringLength(c_Int32 maxStringLength);
```

- `c_Int32 maxStringLength() const;`

  Gets the maximum acceptable string length, inclusive.

- `void maxStringLength(c_Int32 maxStringLength);`

  Sets the maximum acceptable string length, inclusive.

  Parameters

  maxStringLength The maximum acceptable string length.

  Throws

  ccOCRDictionaryDefs::BadParams
  maxStringLength < 1

  Notes

  This parameter has no effect if **fixedLengthFielding()** returns true.

  The **run()** function throws *ccOCRDictionaryDefs::BadParams* if *minStringLength* > *maxStringLength*, and **fixedLengthFielding()** returns false.

maxFieldFirstIndex

```
c_Int32 maxFieldFirstIndex() const;

void maxFieldFirstIndex(c_Int32 maxFirstIndex);
```

- `c_Int32 maxFieldFirstIndex() const;`

  Gets the maximum allowable start position of a subsequence of the fielding vector to be used.

- `void maxFieldFirstIndex(c_Int32 maxFirstIndex);`

  Sets the maximum allowable start position of a subsequence of the fielding vector to be used.

  Parameters

  maxFirstIndex The maximum allowable start position of a subsequence of the fielding vector to be used.

Throws

ccOCRDictionaryDefs::BadParams
maxFirstIndex < 0

Notes

For variable-length fielding, the fielding subsequences to be considered must start at a position no greater than this index.

This parameter has no effect if **fixedLengthFielding()** returns true.

minFieldLastIndex

```
c_Int32 minFieldLastIndex() const;

void minFieldLastIndex(c_Int32 minLastIndex);
```

- `c_Int32 minFieldLastIndex() const;`

Gets the minimum allowable inclusive end position of a subsequence of the fielding vector to be used.

- `void minFieldLastIndex(c_Int32 minLastIndex);`

Sets the minimum allowable inclusive end position of a subsequence of the fielding vector to be used.

Parameters

minLastIndex The minimum allowable inclusive end position of a subsequence of the fielding vector to be used.

Throws

ccOCRDictionaryDefs::BadParams
minLastIndex < 0

Notes

For variable-length fielding, the fielding subsequences to be considered must end at a position no smaller than this index.

This parameter has no effect if **fixedLengthFielding()** returns true.

The **run()** function throws *ccOCRDictionaryDefs::BadParams* if **fixedLengthFielding()** == false and **minFieldLastIndex()** >= **fielding.size()**

ignoreFailingPrefixSuffix

```
bool ignoreFailingPrefixSuffix() const;

void ignoreFailingPrefixSuffix(bool ignore);
```

- `bool ignoreFailingPrefixSuffix() const;`

  Gets whether to ignore failing prefix and suffix when supporting variable string length.

- `void ignoreFailingPrefixSuffix(bool ignore);`

  Sets whether to ignore failing prefix and suffix when supporting variable string length.

  Parameters

  ignore Whether to ignore failing prefix and suffix when supporting variable string length.

  Notes

  A failing position refers to one whose primary character is *ccCharCode::eUnknown*.

  This parameter has no effect if **fixedLengthFielding()** returns true.

isGoodMetric

```
ccOCRDictionaryDefs::IsGoodMetric isGoodMetric() const;

void isGoodMetric(
  ccOCRDictionaryDefs::IsGoodMetric isGoodMetric);
```

- `ccOCRDictionaryDefs::IsGoodMetric isGoodMetric() const;`

  Gets which matches (only primary, or either primary or secondary) are considered to be good.

- `void isGoodMetric(`
  `ccOCRDictionaryDefs::IsGoodMetric isGoodMetric);`

  Sets which matches (only primary, or either primary or secondary) are considered to be good.

  Parameters

  isGoodMetric The matches that are considered to be good.

  Throws

  ccOCRDictionaryDefs::BadParams
  isGoodMetric is not a valid enum value.

Operators

operator==

```
bool operator==(
  const ccOCRDictionaryFieldingRunParams& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

operator!=

```
bool operator!=(
  const ccOCRDictionaryFieldingRunParams& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for inequality.

ccOCRDictionaryFieldingRunParams

ccOCRDictionaryPositionFielding

```
#include <ch_cvl/ocrfldng.h>

class ccOCRDictionaryPositionFielding;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class specifies the set of acceptable characters at a position.

Notes

There are two independent methods for specifying the acceptable characters; that is, via a bit field of the *ccOCRDictionaryDefs::Fielding* enum and via listing all allowed individual characters. These two methods do not interact with each other. The OCR Dictionary Fielding tool uses the union of all characters from the two methods.

Constructors/Destructors

ccOCRDictionaryPositionFielding

```
ccOCRDictionaryPositionFielding(
  ccOCRDictionaryDefs::Fielding fielding =
  ccOCRDictionaryDefs::kDefaultFielding);

ccOCRDictionaryPositionFielding(
  const ccOCRDictionaryPositionFielding& rhs);
```

- ```
  ccOCRDictionaryPositionFielding(
    ccOCRDictionaryDefs::Fielding fielding =
    ccOCRDictionaryDefs::kDefaultFielding);
  ```

  Constructs a default object.

  Parameters

  fielding The fielding value.
  ```

- ```
  ccOCRDictionaryPositionFielding(
    const ccOCRDictionaryPositionFielding& rhs);
  ```

  Copy constructor.

  Parameters

  rhs The source of the copy.

ccOCRDictionaryPositionFielding

```
~ccOCRDictionaryPositionFielding();
```

Destructor.

Public Member Functions

fieldingBits

```
c_UInt32 fieldingBits() const;

void fieldingBits(c_UInt32 fieldingBits);
```

- ```
  c_UInt32 fieldingBits() const;
  ```

  Gets the fielding bit field.

- ```
  void fieldingBits(c_UInt32 fieldingBits);
  ```

  Sets the fielding bit field.

  Parameters

  fieldingBits The fielding bit field.

  Throws

  ccOCRDictionaryDefs::BadParams
  fieldingBits is not a bitwise OR of values from *ccOCRDictionaryDefs::Fielding*.

  Notes

  This does not interact with **fieldingCharacters()**.
  ```

fieldingCharacters

```
void fieldingCharacters(
  const cmStd vector<c_Char32>& characters);

const cmStd vector<c_Char32>& fieldingCharacters() const;
```

- ```
  void fieldingCharacters(
    const cmStd vector<c_Char32>& characters);
  ```

  Sets the fielding characters.

  Parameters

  characters The fielding characters to be set.

  Throws

  ccOCRDictionaryDefs::BadParams
  characters contain duplicates, or
  any element of *characters* is *ccCharCode::eUnknown*.

  Notes

  This is a straight setter. It does not attempt to collapse the specified characters into a minimum equivalent set of characters if the specified characters contain any special values such as *ccCharCode::eAnyCharacter*.

- ```
  const cmStd vector<c_Char32>& fieldingCharacters() const;
  ```

  Gets the fielding characters.

  Notes

  This is a straight getter. It does not attempt to collapse the set of fielding characters even if they contain special values such as *ccCharCode::eAnyCharacter*.

  This does not interact with **fieldingBits()**.

allFieldingCharacters

```
cmStd vector<c_Char32> allFieldingCharacters() const;
```

Gets the minimum equivalent set of fielding characters.

Notes

This attempts to collapse the set of fielding characters as much as possible to throw away redundant ones. For example, if **fieldingCharacters()** contained the special value *ccCharCode::eAnyNonspaceCharacter* and a value 0x20 (space) in addition to other values, **allFieldingCharacters()** would return a vector with a single element of the value *ccCharCode::eAnyCharacter*.

This does not interact with **fieldingBits()**.

fielding

```
void fielding(
  c_UInt32 fieldingBits,
  const cmStd vector<c_Char32>& fieldingCharacters);
```

Sets all allowed characters via both bit field and characters.

Parameters

fieldingBits The fielding bits.

fieldingCharacters
The fielding characters.

Notes

The fielding bits and characters are maintained independently. The tool uses their union.

multiCharacter

```
ccOCRDictionaryCharMulti multiCharacter() const;
```

Gets a multicharacter representation of the union of all characters allowed for this fielding, via either fielding bits or fielding characters.

Notes

The individual characters in the returned object are ordered in ascending order of the character code.

If **isAnyCharacter()** or **isAnyNonspaceCharacter()** returns true, the returned multicharacter object will contain exactly one element: **ccOCRDictionaryChar(***ccCharCode::eAnyCharacter***)** or **ccOCRDictionaryChar(***ccCharCode::eAnyNonspaceCharacter***)**, as appropriate.

isEmpty

```
bool isEmpty() const;
```

Gets whether this fielding allows no character.

Notes

This returns true if both of the following are true:

- **fieldingBits()** == 0
- **fieldingCharacters().size()** == 0

isSingular

```
bool isSingular() const;
```

Gets whether this fielding allows only a single character.

isAnyCharacter

```
bool isAnyCharacter() const;
```

Returns whether this fielding allows any character.

Notes

This checks the union of the bit field and the characters.

isAnyCharacter() and **isAnyNonspaceCharacter()** cannot both be true at the same time. Specifically, **isAnyNonspaceCharacter()** being true means the fielding does not allow the space character 0x20, while **isAnyCharacter()** being true allows it.

isAnyNonspaceCharacter

```
bool isAnyNonspaceCharacter() const;
```

Returns whether this fielding allows any nonspace character.

Notes

This checks the union of the bit field and the characters.

isAnyCharacter() and **isAnyNonspaceCharacter()** cannot both be true at the same time. Specifically, **isAnyNonspaceCharacter()** being true means the fielding does not allow the space character 0x20, while **isAnyCharacter()** being true allows it.

Operators

operator==

```
bool operator==(
  const ccOCRDictionaryPositionFielding& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCRDictionaryPositionFielding

operator!=

```
bool operator!=(
  const ccOCRDictionaryPositionFielding& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

operator=

```
ccOCRDictionaryPositionFielding& operator=(
  const ccOCRDictionaryPositionFielding& rhs);
```

Assignment operator.

Parameters

rhs The object to assign to this one.

ccOCRDictionaryResult

```
#include <ch_cvl/ocrdcbas.h>

class ccOCRDictionaryResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRDictionaryResult contains the results from applying the OCR Dictionary Fielding tool to the output from the OCR Classifier tool. The result includes information about each position of the input string and the result string.

Constructors/Destructors

ccOCRDictionaryResult

```
ccOCRDictionaryResult();
```

Construct a default dictionary result object. **isComputed()** returns *false* on a default-constructed **ccOCRDictionaryResult**.

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Reset this object to its default state.

isComputed

```
bool isComputed() const;
```

Returns *true* if this **ccOCRDictionaryResult** contains valid data, *false* otherwise.

Notes

All other getters will throw *ccOCRDictionaryDefs::NotComputed* if **isComputed()** returns *false*.

isGoodMetric `ccOCRDictionaryDefs::IsGoodMetric isGoodMetric() const;`

Gets whether primary and/or secondary matches are considered good.

isGood `bool isGood() const;`

If **isGoodMetric()**==*ccOCRDictionaryDefs::ePrimaryMatchesAreGood,* this function returns *true* under the following conditions:

- All result string positions have a status of *ccOCRDictionaryDefs::ePrimary* or *ccOCRDictionaryDefs::ePrimarySwap.*
- All input string positions have a status of *ccOCRDictionaryDefs::ePrimary*, *ccOCRDictionaryDefs::ePrimarySwap,* or *ccOCRDictionaryDefs::eDeletedIgnored*.

If **isGoodMetric()**==*ccOCRDictionaryDefs::ePrimaryAndSecondaryMatchesAreGood*, this function returns *true* under the following conditions:

- All result string positions have a status of *ccOCRDictionaryDefs::ePrimary*, *ccOCRDictionaryDefs::ePrimarySwap*, *ccOCRDictionaryDefs::eSecondary*, or *ccOCRDictionaryDefs::eSecondarySwap*.
- All input string positions have a status of *ccOCRDictionaryDefs::ePrimary*, *ccOCRDictionaryDefs::ePrimarySwap*, *ccOCRDictionaryDefs::eSecondary*, *ccOCRDictionaryDefs::eSecondarySwap*, or *ccOCRDictionaryDefs::eDeletedIgnored*.

inputStringMulti
```
const ccOCRDictionaryStringMulti&
   inputStringMulti() const;
```

Returns a reference to the input string used in the run that produced this result.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

inputString `ccOCRDictionaryString inputString() const;`

Returns the input string used in the run that produced this result. This function can only be used in cases where the input string includes a single character only.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

ccOCRDictionaryDefs::NotExpressible
ccOCRDictionary::inputStringMulti().isSingular() is false.

inputStringLineStatus

```
ccOCRDictionaryDefs::PositionStatus
   inputStringLineStatus() const;
```

Returns the overall status for the input string. The overall status is defined as the *lowest* status grade among all the individual position statuses. If there are no position statuses (the vector returned by the first overload of **inputStringPositionStatus()** is empty), then this function returns *ccOCRDictionaryDefs::eNoMatch*.

Throws
ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

inputStringPositionStatus

```
const cmStd vector<ccOCRDictionaryDefs::PositionStatus>&
   inputStringPositionStatus() const;

ccOCRDictionaryDefs::PositionStatus
   inputStringPositionStatus(c_Int32 pos) const;
```

- ```
  const cmStd vector<ccOCRDictionaryDefs::PositionStatus>&
     inputStringPositionStatus() const;
  ```

  Returns the position statuses for each position in the input string.

  Throws
  ccOCRDictionaryDefs::NotComputed
  This **ccOCRDictionaryResult** does not contain valid computed results.

- ```
  ccOCRDictionaryDefs::PositionStatus
     inputStringPositionStatus(c_Int32 pos) const;
  ```

  Returns the position status for the specified position in the input string.

  Parameters
  pos The string position.

  Throws
  ccOCRDictionaryDefs::NotComputed
  This **ccOCRDictionaryResult** does not contain valid computed results.

ccOCRDictionaryDefs::BadParam
pos is less than 0 or greater than or equal to the number of position statuses (**inputStringPositionStatus().size()**).

resultString

```
const ccOCRDictionaryString& resultString() const
```

Returns a reference to the result string created by the OCR Dictionary Fielding tool.

Throws
ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

resultStringLineStatus

```
ccOCRDictionaryDefs::PositionStatus
   resultStringLineStatus() const;
```

Returns the overall status for the result string. The overall status is defined as the *lowest* status grade among all the individual position statuses. If there are no position statuses (the vector returned by the first overload of **resultStringPositionStatus()** is empty), then this function returns *ccOCRDictionaryDefs::eNoMatch*.

Throws
ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

resultStringPositionStatus

```
const cmStd vector<ccOCRDictionaryDefs::PositionStatus>&
   resultStringPositionStatus() const;

ccOCRDictionaryDefs::PositionStatus
   resultStringPositionStatus(c_Int32 pos) const;
```

-
```
const cmStd vector<ccOCRDictionaryDefs::PositionStatus>&
   resultStringPositionStatus() const;
```

Returns the position statuses for each position in the result string.

Throws
ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

- ```
  ccOCRDictionaryDefs::PositionStatus
     resultStringPositionStatus(c_Int32 pos) const;
  ```

  Returns the position status for the specified position in the result string.

  Parameters

  pos The string position.

  Throws

  ccOCRDictionaryDefs::NotComputed
  This **ccOCRDictionaryResult** does not contain valid computed results.

  ccOCRDictionaryDefs::BadParam
  pos is less than 0 or greater than or equal to the number of position statuses (**resultStringPositionStatus().size()**).

numIgnoredPrefixCharacters

```
c_Int32 numIgnoredPrefixCharacters() const;
```

Return the number of prefix characters that are ignored from the beginning of the string.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

numIgnoredSuffixCharacters

```
c_Int32 numIgnoredSuffixCharacters() const;
```

Return the number of suffix characters that are ignored from the end of the string.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

numPrimaryCharacters

```
c_Int32 numPrimaryCharacters(
   bool includingSwaps = true) const;
```

Returns the number of primary characters in the result string. You can optionally include the number of their swap characters as well.

Parameters

includingSwaps Specify *true* to return the number of primary characters and their swaps, *false* to return just the number of primary characters.
```

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

numSecondaryCharacters

```
c_Int32 numSecondaryCharacters(
   bool includingSwaps = true) const;
```

Returns the number of secondary characters in the result string. You can optionally include the number of their swap characters as well.

Parameters

includingSwaps Specify *true* to return the number of secondary characters and their swaps, *false* to return just the number of secondary characters.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

inputStringIndexFromResultStringIndex

```
c_Int32 inputStringIndexFromResultStringIndex(
   c_Int32 resultStringIndex) const;
```

Returns the character index in the input string that corresponds to the specified character index in the result string.

Notes

If you specify a result string character index that refers to an inserted character, this function returns -1.

Parameters

resultStringIndex The result string character index.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

ccOCRDictionaryDefs::BadParams
resultStringIndex is less than 0 or greater than or equal to the number of position statuses (**resultStringPositionStatus().size()**).

fieldFirstIndex `c_Int32 fieldFirstIndex() const;`

Gets the position in the fielding where the fielding was matched to the input string. This parameter is only useful for variable length fielding results where the matched fielding can be a subset of the user-specified fielding.

Notes

This value is always 0 for fixed length fielding results.

This value is always greater than or equal to 0 for variable length fielding results.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

Operators

operator== `bool operator==(const ccOCRDictionaryResult& rhs) const;`

Returns true if this **ccOCRDictionaryResult** is equal to the supplied one, false otherwise.

Parameters

rhs The **ccOCRDictionaryResult** to compare to this one.

operator!= `bool operator!=(const ccOCRDictionaryResult& rhs) const`

Returns false if this **ccOCRDictionaryResult** is equal to the supplied one, true otherwise.

Parameters

rhs The **ccOCRDictionaryResult** to compare to this one.

ccOCRDictionaryResult

ccOCRDictionaryResultSet

```
#include <ch_cvl/ocrdcbas.h>

class ccOCRDictionaryResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRDictionaryResultSet contains a collection of all results from applying the OCR Dictionary Fielding tool to an input string.

Constructors/Destructors

ccOCRDictionaryResultSet

```
ccOCRDictionaryResultSet();
```

Construct a default dictionary result object. **isComputed()** returns *false* on a default-constructed **ccOCRDictionaryResultSet**.

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

reset

```
void reset();
```

Reset this object to its default state.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns *true* if this **ccOCRDictionaryResultSet** contains valid data, *false* otherwise.

Notes

All other getters will throw *ccOCRDictionaryDefs::NotComputed* if **isComputed()** returns *false*.

results

```
const cmStd vector<ccOCRDictionaryResult>& results() const;
```

Returns a reference to a vector containing all of the results objects.

Notes

The results are sorted in descending order of quality rating.

Throws

ccOCRDictionaryDefs::NotComputed
This **ccOCRDictionaryResult** does not contain valid computed results.

Operators

operator==

```
bool operator==(const ccOCRDictionaryResultSet& rhs) const;
```

Returns true if this **ccOCRDictionaryResultSet** is equal to the supplied one, false otherwise.

Parameters

rhs The **ccOCRDictionaryResultSet** to compare to this one.

operator!=

```
bool operator!=(const ccOCRDictionaryResultSet& rhs) const;
```

Returns false if this **ccOCRDictionaryResultSet** is equal to the supplied one, true otherwise.

Parameters

rhs The **ccOCRDictionaryResultSet** to compare to this one.

ccOCRDictionaryString

```
#include <ch_cvl/ocrdcbas.h>

class ccOCRDictionaryString;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRDictionaryString represents a string of characters used by the OCR Dictionary Fielding tool (both for input and output). **ccOCRDictionaryString** contains a vector of **ccOCRDictionaryChar** objects, one for each string position.

For strings containing multicharacters, use **ccOCRDictionaryStringMulti**.

Constructors/Destructors

ccOCRDictionaryString

```
ccOCRDictionaryString(
   const cmStd vector<c_Char32>& characters, double score);

ccOCRDictionaryString(const ccCvlString& characters,
   double score);

explicit ccOCRDictionaryString(
  const cmStd vector<ccOCRDictionaryChar>& characters);
```

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

- ```
  ccOCRDictionaryString(
    const cmStd vector<c_Char32>& characters, double score);
  ```

  Constructs a **ccOCRDictionaryString** from the supplied vector of characters. All characters are assigned the supplied score value.

  Parameters

  characters The characters to place in the **ccOCRDictionaryString**.
  ```

score The score, which must be in the range 0.0 through 1.0, to assign to the characters.

Throws

ccOCRDictionaryDefs::BadParams

score is less than 0.0 or greater than 1.0.

- `ccOCRDictionaryString(const ccCvlString& characters, double score);`

Constructs a **ccOCRDictionaryString** from the supplied **ccCVLString** object. All characters are assigned the supplied score value.

Parameters

characters The string containing the characters to place in the **ccOCRDictionaryString**.

score The score, which must be in the range 0.0 through 1.0, to assign to the characters.

Throws

ccOCRDictionaryDefs::BadParams

score is less than 0.0 or greater than 1.0.

- `explicit ccOCRDictionaryString( const cmStd vector<ccOCRDictionaryChar>& characters);`

Constructs a **ccOCRDictionaryString** from the supplied vector of **ccOCRDictionaryChar** objects. The supplied characters' scores are not changed.

Public Member Functions

characters

```
const cmStd vector<ccOCRDictionaryChar>& characters()
   const;

void characters(
   const cmStd vector<ccOCRDictionaryChar>& c);

void characters(const cmStd vector<c_Char32>& characters,
   double score);

void characters(const ccCvlString& characters,
   double score);
```

- ```
  const cmStd vector<ccOCRDictionaryChar>& characters()
     const;
  ```

  Returns a reference to the characters in this **ccOCRDictionaryString**.

- ```
  void characters(
     const cmStd vector<ccOCRDictionaryChar>& c);
  ```

  Sets the characters in this **ccOCRDictionaryString** to be the supplied vector of **ccOCRDictionaryChar**. The characters' scores are not changed.

  Parameters

  c The characters to set.

  Throws

  ccOCRDictionaryDefs::BadParams
  score is less than 0.0 or greater than 1.0.

- ```
  void characters(const cmStd vector<c_Char32>& characters,
     double score);
  ```

  Sets the characters in this **ccOCRDictionaryString** to be the supplied vector of characters. All characters are assigned the supplied score value.

  Parameters

  characters The characters to set.

  score The score for the characters, in the range 0.0 through 1.0.

  Throws

  ccOCRDictionaryDefs::BadParams
  score is less than 0.0 or greater than 1.0.
  ```

- `void characters(const ccCvlString& characters, double score);`

  Sets the characters in this **ccOCRDictionaryString** to be the supplied **ccCVLString** object. All characters are assigned the supplied score value.

  Parameters

  characters The characters to set.

  score The score for the characters, in the range 0.0 through 1.0.

  Throws

  ccOCRDictionaryDefs::BadParams
  score is less than 0.0 or greater than 1.0.

size `c_Int32 size() const;`

Returns the number of characters in this **ccOCRDictionaryString**.

isEmpty `bool isEmpty() const;`

Returns *true* if there are no characters in this **ccOCRDictionaryString**, *false* otherwise.

Operators

operator ccCvlString

`operator ccCvlString() const;`

Returns the **ccCVLString** representation of this string. If either **ccOCRDictionaryChar::char()** or **ccOCRDictionaryChar::wchar_t()** throws an error for any character in the string, this function will also throw that error.

Throws

ccCharCodeDefs::NotRepresentable
A character in this string cannot be represented as a **ccCvlChar**.

---

operator[]

```
const ccOCRDictionaryChar& operator[](c_Int32 index) const;

ccOCRDictionaryChar& operator[](c_Int32 index);
```

- `const ccOCRDictionaryChar& operator[](c_Int32 index) const;`

  Return a **const** reference to the given character in the string.

  Parameters

  index The character to return.

Throws

ccOCRDictionaryDefs::BadParam

index is less than 0 or greater than or equal to the number of characters in the string.

- `ccOCRDictionaryChar& operator[](c_Int32 index);`

Return a non-**const** reference to the given character in the string.

Parameters

index The character to return.

Throws

ccOCRDictionaryDefs::BadParam

index is less than 0 or greater than or equal to the number of characters in the string.

operator== `bool operator==(const ccOCRDictionaryString& rhs) const;`

Returns *true* if this **ccOCRDictionaryString** is equal to the supplied one, *false* otherwise.

Parameters

rhs The **ccOCRDictionaryString** to compare to this one.

operator!= `bool operator!=(const ccOCRDictionaryString& rhs) const;`

Returns *false* if this **ccOCRDictionaryString** is equal to the supplied one, *true* otherwise.

Parameters

rhs The **ccOCRDictionaryString** to compare to this one.

ccOCRDictionaryString

ccOCRDictionaryStringMulti

```
#include <ch_cvl/ocrdcbas.h>

class ccOCRDictionaryStringMulti;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccOCRDictionaryStringMulti represents a string of characters used by the OCR Dictionary Fielding tool (both for input and output) where each string position contain a multicharacter. **ccOCRDictionaryStringMulti** contains a vector of **ccOCRDictionaryCharMulti** objects, one for each string position.

Constructors/Destructors

ccOCRDictionaryStringMulti

```
explicit ccOCRDictionaryStringMulti(const
   cmStd vector<ccOCRDictionaryCharMulti>& characters);

explicit ccOCRDictionaryStringMulti(
   const ccOCRDictionaryString& characters);
```

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

- ```
  explicit ccOCRDictionaryStringMulti(const
     cmStd vector<ccOCRDictionaryCharMulti>& characters);
  ```

  Constructs a **ccOCRDictionaryStringMulti** from the supplied vector of **ccOCRDictionaryCharMulti** objects.

  Parameters

characters	The characters from which to construct this **ccOCRDictionaryStringMulti**.
  ```

- ```
  explicit ccOCRDictionaryStringMulti(
     const ccOCRDictionaryString& characters);
  ```

  Constructs a **ccOCRDictionaryStringMulti** from the supplied vector of **ccOCRDictionaryString**. Each **ccOCRDictionaryChar** in the supplied string is converted to a **ccOCRDictionaryCharMulti** with a single character.

  Parameters

  characters The characters from which to construct this **ccOCRDictionaryStringMulti**.

Public Member Functions

characters

```
const cmStd vector<ccOCRDictionaryCharMulti>&
   characters() const;

void characters(const
   cmStd vector<ccOCRDictionaryCharMulti>& c);
```

- ```
  const cmStd vector<ccOCRDictionaryCharMulti>&
     characters() const;
  ```

  Returns a reference to the characters in this **ccOCRDictionaryStringMulti**.

- ```
  void characters(const
     cmStd vector<ccOCRDictionaryCharMulti>& c);
  ```

  Sets the characters in this **ccOCRDictionaryStringMulti** from the supplied vector of multicharacters.

  Parameters

  c The characters to set.

isSingular

```
bool isSingular() const;
```

Returns *true* if each character in this object contains exactly one character, false otherwise.

Notes

This function returns *true* if the string itself is empty (**size()** is 0).

size

```
c_Int32 size() const;
```

Gets the number of multicharacters in this string.
```

dictionaryString

```
ccOCRDictionaryString dictionaryString(
   bool throwIfAmbiguous = false) const;

void dictionaryString(const ccOCRDictionaryString& s);
```

- ```
  ccOCRDictionaryString dictionaryString(
     bool throwIfAmbiguous = false) const;
  ```

  Return a **ccOCRDictionaryString** that is made up of

  - A character position's primary character, if the position has multiple characters.
  - A character position's character, if the position has only a single character.

  If you supply a value of *true* for *throwIfAmbiguous*, then this function will throw an error if any character position has multiple characters.

  Parameters

  throwIfAmbiguous

  Specify a value of *true*, to force an error if any character positions have multiple characters.

  Throws

  ccCharCodeDefs::NotExpressible

  throwIfAmbiguous is *true* and at least one character position in this object has multiple characters or *throwIfAmbiguous* is false and at least one character position has multiple characters and at least one character position has no primary character.

- ```
  void dictionaryString(const ccOCRDictionaryString& s);
  ```

  Sets this **ccOCRDictionaryStringMulti**'s characters using the supplied **ccOCRDictionaryString** object. Each character position will have a single character.

  Parameters

  s The string to set.

Operators

operator[]

```
const ccOCRDictionaryCharMulti& operator[](
   c_Int32 index) const;

ccOCRDictionaryCharMulti& operator[](c_Int32 index);
```

- ```
  const ccOCRDictionaryCharMulti& operator[](
     c_Int32 index) const;
  ```

  Return a **const** reference to the given character in the string.

  Parameters

  index The character to return.

  Throws

  ccOCRDictionaryDefs::BadParam

  index is less than 0 or greater than or equal to the number of characters in the string.

- ```
  ccOCRDictionaryCharMulti& operator[](c_Int32 index);
  ```

  Return a non-**const** reference to the given character in the string.

  Parameters

  index The character to return.

  Throws

  ccOCRDictionaryDefs::BadParam

  index is less than 0 or greater than or equal to the number of characters in the string.

operator==

```
bool operator==(const ccOCRDictionaryStringMulti& rhs)
   const;
```

Returns *true* if this **ccOCRDictionaryStringMulti** is equal to the supplied one, *false* otherwise.

Parameters

rhs The **ccOCRDictionaryStringMulti** to compare to this one.

operator!=

```
bool operator!=(const ccOCRDictionaryStringMulti& rhs)
   const;
```

Returns *false* if this **ccOCRDictionaryStringMulti** is equal to the supplied one, *true* otherwise.

Parameters

rhs	The **ccOCRDictionaryStringMulti** to compare to this one.

ccOCRDictionaryStringMulti

ccOCSwapChar

```
#include <ch_cvl/ocswapch.h>

class ccOCSwapChar;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class defines a single group of swap characters. Characters within the group are swappable with each other.

Constructors/Destructors

ccOCSwapChar

```
ccOCSwapChar();

explicit ccOCSwapChar(const cmStd vector<c_Char32>&
  characterCodes);

ccOCSwapChar(c_Char32 c1, c_Char32 c2);
```

- `ccOCSwapChar();`

  Constructs an object.

- `explicit ccOCSwapChar(const cmStd vector<c_Char32>& characterCodes);`

  Constructs an object with the specified group of swap characters.

  Parameters

  characterCodes The character codes of swappable characters.

  Throws

  ccOCSwapCharDefs::BadParams

  characterCodes contains any duplicate character codes, or **characterCodes.size()** < 2;
  or
  any of the character codes are Cognex-reserved character

codes (use the **ccCharCode::isReserved()** function to determine whether it is a Cognex-reserved character code and refer to <*ch_cvl/charcode.h*>).

- `ccOCSwapChar(c_Char32 c1, c_Char32 c2);`

Constructs an object with the specified group of swap characters.

Parameters

c1 The character code of one of the swappable characters.

c2 The character code of the other swappable character.

Throws

ccOCSwapCharDefs::BadParams
c1 == *c2*, or
any of the character codes are Cognex-reserved character codes (use the **ccCharCode::isReserved()** function to determine whether it is a Cognex-reserved character code and refer to <*ch_cvl/charcode.h*>).

Notes

Compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

characters

```
const cmStd vector<c_Char32>& characters() const;

void characters(const cmStd vector<c_Char32>&
  characterCodes);

void characters(c_Char32 c1, c_Char32 c2);
```

- `const cmStd vector<c_Char32>& characters() const;`

Gets members of the swap character group.

Throws

ccOCSwapCharDefs::BadParams
characterCodes.size() < 2, or
any of the character codes are Cognex-reserved character codes (use the **ccCharCode::isReserved()** function to determine whether it is a Cognex-reserved character code and refer to <*ch_cvl/charcode.h*>)

- `void characters(const cmStd vector<c_Char32>& characterCodes);`

  Sets members of the swap character group.

  Parameters

  characterCodes

  Character codes of characters to be set as members of the swap character group.

  Throws

  ccOCSwapCharDefs::BadParams

  characterCodes contains any duplicate character codes, or *c1 == c2*;
  or
  characterCodes.size() < 2, or
  any of the character codes are Cognex-reserved character codes (use the **ccCharCode::isReserved()** function to determine whether it is a Cognex-reserved character code and refer to <*ch_cvl/charcode.h*>).

- `void characters(c_Char32 c1, c_Char32 c2);`

  Sets members of the swap character group.

  Parameters

  c1 — The first character of the character pair to be set as member of the swap character group.

  c2 — The second character of the character pair to be set as member of the swap character group

  Throws

  ccOCSwapCharDefs::BadParams

  characterCodes contains any duplicate character codes, or *c1 == c2*;
  or
  characterCodes.size() < 2, or
  any of the character codes are Cognex-reserved character codes (use the **ccCharCode::isReserved()** function to determine whether it is a Cognex-reserved character code and refer to <*ch_cvl/charcode.h*>).

Notes

The setters set, instead of append, to the group.

The setters sort the character codes in ascending order.

Setters throw an exception if *characterCodes* contains any duplicate character codes, or if *c1* == *c2*.

Operators

degen

```
bool degen() const;
```

Returns whether this group of swap character is degenerate, that is, contains less than two members.

contains

```
bool contains(c_Char32 c) const;
```

Returns whether *c* is among this group of swap characters.

Parameters

c	The character whose swap group membership is to be investigated.

operator==

```
bool operator==(const ccOCSwapChar& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCSwapChar& rhs) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for inequality.

ccOCSwapCharSet

```
#include <ch_cvl/ocswapch.h>

class ccOCSwapCharSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class defines a set of swap character groups. Characters within each group are swappable with each other.

Constructors/Destructors

ccOCSwapCharSet

```
ccOCSwapCharSet(const cmStd vector<ccOCSwapChar>&
  swapCharacters = cmStd vector<ccOCSwapChar>());

ccOCSwapCharSet(const ccOCSwapCharSet& rhs);
```

- ```
  ccOCSwapCharSet(const cmStd vector<ccOCSwapChar>&
    swapCharacters = cmStd vector<ccOCSwapChar>());
  ```

  Constructs an object with the specified set of swap character groups.

  Parameters

  swapCharacters The set of swap character groups.

  Throws

  ccOCSwapCharDefs::BadParams
  by setter if **degen()** is true for any element of *swapCharacters*.

- ```
  ccOCSwapCharSet(const ccOCSwapCharSet& rhs);
  ```

  Copy constructor.

  Parameters

  rhs The source of the copy.

ccOCSwapCharSet

```
~ccOCSwapCharSet();
```

Destructor.

Public Member Functions

swapCharacters

```
const cmStd vector<ccOCSwapChar>& swapCharacters() const;

void swapCharacters(const cmStd vector<ccOCSwapChar>&
  swapCharacters);
```

- `const cmStd vector<ccOCSwapChar>& swapCharacters() const;`

  Gets all swap character groups in this set.

- `void swapCharacters(const cmStd vector<ccOCSwapChar>& swapCharacters);`

  Sets all swap character groups in this set.

  Parameters

  swapCharacters All swap character groups in this set.

  Throws

  ccOCSwapCharDefs::BadParams
  degen() is true for any element of *swapCharacters*.

  Notes

  By transitivity, all groups that share a same member character with one another are merged into a single group.

  Calling the setter with an empty vector creates an empty set. An empty set means that there are no swappable characters.

  The setter sorts the character codes within each group in ascending order.

add

```
void add(const ccOCSwapChar& swapCharacter);
```

Adds a group of swap characters to this set.

Parameters

swapCharacter The group of swap characters to be added.

Throws

ccOCSwapCharDefs::BadParams
swapCharacter.degen() is true.

Notes

By transitivity, if any character in the group already exists in an existing group in the set, the new group is merged with that existing group.

getSwapCharacter

```
void getSwapCharacter(
  c_Char32 c,
  ccOCSwapChar& swapCharacter) const;
```

Returns the group of swap characters that contains *c*.

Parameters

c	The character to search for.
swapCharacter	The swap character.

Notes

Returns a default empty **ccOCSwapChar** object if no group in this set contains *c*.

canSwap

```
bool canSwap(c_Char32 c1, c_Char32 c2) const;
```

Returns whether *c1* and *c2* exist in the same swap character group within this set.

Parameters

c1	The first character to check.
c2	The second character to check.

contains

```
bool contains(c_Char32 c) const;
```

Returns whether any swap character group in this set contains *c*.

Parameters

c	The character to check.

Operators

operator==

```
bool operator==(const ccOCSwapCharSet& rhs) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

rhs	The object with which to compare for equality.

ccOCSwapCharSet

operator!= `bool operator!=(const ccOCSwapCharSet& rhs) const;`

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

rhs The object with which to compare for inequality.

operator= `ccOCSwapCharSet& operator=(const ccOCSwapCharSet& rhs);`

Assignment operator.

Parameters

rhs The object to assign to this one.

ccOCVDefs

```
#include <ch_cvl/ocv.h>

class ccOCVDefs;
```

A name space that holds enumerations and constants used with the OCV tool.

Enumerations

CharStatus

```
enum CharStatus ;
```

This enumeration defines verification status results for characters.

Value	Meaning
eVerified	The character passes verification. The matching score is greater than the accept threshold, and the difference between the matching score and the highest scoring confusing character is greater than the confidence threshold.
eConfused	The matching score for the character is greater than the accept threshold, however the difference between the matching score and the highest scoring confusing character is not greater than the confidence threshold.
eFailed	The character fails verification. Its matching score is less than the accept threshold.

DrawFlags

```
enum DrawFlags;
```

This enumeration determines which graphics are included when diagnostics are generated.

Value	Meaning
eDrawArrangement	Draw the final arrangement pose.
eDrawLines	Draw the final poses of each line.
eDrawPositions	Draw the final poses of each character.
eDrawArrangementBox	Draw a bounding box when drawing the final arrangement pose.

Value	Meaning
eDrawLineBoxes	Draw a bounding box when drawing the final line pose.
eDrawPositionBoxes	Draw a bounding box when drawing the final character pose.

ccOCVLineResult

```
#include <ch_cvl/ocv.h>

class ccOCVLineResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class captures information about the OCV tool's verification results for one line within a line arrangement. You should not create a **ccOCVLineResult** directly.

Constructors/Destructors

ccOCVLineResult

```
ccOCVLineResult();
```

Constructs a result object with a failed status and no character position results.

Operators

operator==

```
bool operator==(const ccOCVLineResult& other) const;
```

Returns true if this object is equal to the specified line result object.

Parameters

other The line result object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVLineResult& other) const;
```

Returns true if this object is not equal to the specified line result object.

Parameters

other The line result object with which to compare for inequality.

Public Member Functions

lineIndex

```
c_Int32 lineIndex() const;
```

Returns the index of the line within the arrangement.

clientPose

```
const cc2Xform& clientPose() const;
```

Returns the pose of the line in client coordinates. If no characters in this line were found, the returned pose is the expected pose of the line, based on the positions of the neighboring lines within the arrangement.

arrangementPose

```
const cc2Rigid& arrangementPose() const;
```

Returns the pose of the line in arrangement coordinates. If no characters in this line were found, the returned pose is the expected pose of the line, based on the positions of the neighboring lines within the arrangement.

score

```
double score() const;
```

Returns the overall verification score for the line. This score is the average of the verification scores for all characters in the line that the tool attempted to verify.

numPosVerified

```
c_Int32 numPosVerified() const;
```

Returns the number of character positions reported as verified.

verified

```
bool verified() const;
```

Returns true if all character positions in the line passed verification.

posResults

```
const cmStd vector<ccOCVPosResult>& posResults() const;
```

Returns the results of all character positions for which the OCV tool has attempted verification.

posResult

```
const ccOCVPosResult& posResult(c_Int32 posIndex) const;
```

Returns the result for the specified character position. If no character was found, the expected position, based on the positions of nearby found characters, is returned.

Parameters

posIndex The index of the character position.

Throws

ccOCVDefs::BadIndex

posIndex is not the index of a character position for which verification was attempted.

ccOCVLineResult

ccOCVLineRunParams

```
#include <ch_cvl/ocv.h>

class ccOCVLineRunParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class defines the run-time parameters used by the OCV tool to verify a line of text.

Constructors/Destructors

ccOCVLineRunParams

```
ccOCVLineRunParams();

ccOCVLineRunParams(c_Int32 lineIndex,
  const cmStd vector<ccOCVPosRunParams>& posParams);

ccOCVLineRunParams(c_Int32 lineIndex,
  const ccOCLine& line,
  double acceptThreshold = 0.5,
  double confidenceThreshold = 0.0);
```

- `ccOCVLineRunParams();`

  Constructs a default line parameters object with no character position parameters.

- ```
  ccOCVLineRunParams(c_Int32 lineIndex,
    const cmStd vector<ccOCVPosRunParams>& posParams);
  ```

  Constructs a line run-time parameters object with the specified line index and character position parameters.

  Parameters

lineIndex	The line index for these run-time parameters.
posParams	A set of run-time parameters for each character position within this line.
  ```

Throws

ccOCVDefs::BadIndex

lineIndex is less than 0 or *posParams* contains repeated index values.

- ```
  ccOCVLineRunParams(c_Int32 lineIndex,
    const ccOCLine& line,
    double acceptThreshold = 0.5,
    double confidenceThreshold = 0.0);
  ```

Constructs a line run-time parameters object with the specified line index. The run-time parameters for all character positions in the supplied line are automatically constructed using the supplied accept and confidence thresholds.

Parameters

lineIndex The line index for these run-time parameters.

line The line for which to automatically create run-time position parameters.

acceptThreshold

The accept threshold for all character positions within the supplied line. The accept threshold specifies the matching score below which the character at the line position fails verification.

confidenceThreshold

The confidence threshold for all character positions within the supplied line. If the difference between the matching score for a character position and the highest scoring confusing character is greater than the confidence threshold, the position passes verification; otherwise, the position receives a verification status of confused.

Notes

When the OCV tool runs, it verifies all character positions in the line in sequential order.

Throws

ccOCVDefs::BadIndex

lineIndex is less than 0.

ccOCVDefs::BadParams

acceptThreshold or *confidenceThreshold* is less than 0 or greater than 1.
  ```

Operators

operator==

```
bool operator==(const ccOCVLineRunParams& other) const;
```

Returns true if this object is equal to the specified object.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVLineRunParams& other) const;
```

Returns true if this object is not equal to the specified object.

Parameters

other The object with which to compare for inequality.

Public Member Functions

lineIndex

```
void lineIndex(c_Int32 index);
c_Int32 lineIndex() const;
```

- `void lineIndex(c_Int32 index);`

  Sets the index of the line in a line arrangement for which this run-time parameters object refers.

  Parameters

  index The line index.

  Throws

  ccOCVDefs::BadIndex
  index is less than 0.

- `c_Int32 lineIndex() const;`

  Returns the index of the line in a line arrangement for which this run-time parameters object refers.

ccOCVLineRunParams

posParams

```
void posParams(
  const cmStd vector<ccOCVPosRunParams>& params);

const cmStd vector<ccOCVPosRunParams>& posParams() const;
```

- ```
  void posParams(
    const cmStd vector<ccOCVPosRunParams>& params);
  ```

  Sets the run-time parameters for each of the character positions to be verified in a line.

  Parameters

  params — The run-time parameters for each character position in a line, specified in the order that they should be verified.

  Throws

  ccOCVDefs::BadIndex — *params* contains repeated index values for character positions.

- ```
  const cmStd vector<ccOCVPosRunParams>& posParams() const;
  ```

  Returns the run-time parameters for each of the character positions to be verified in a line.

ccOCVMaxArrangement

```
#include <ch_cvl\ocmax.h>

class ccOCVMaxArrangement;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The **ccOCVMaxArrangement** class represents a spatial arrangement of paragraphs or set of character strings you want the tool to verify.

Constructors/Destructors

ccOCVMaxArrangement

```
ccOCVMaxArrangement();

ccOCVMaxArrangement(
  const cmStd vector<ccOCVMaxParagraph> &paras,
  const cmStd vector<cc2Xform> &paraPoses);

ccOCVMaxArrangement(
  const cmStd vector<ccOCVMaxParagraphPtrh_const> &paras,
  const cmStd vector<cc2Xform> &paraPoses);

ccOCVMaxArrangement(
  const cmStd vector<ccOCVMaxParagraphPtrh> &paras,
  const cmStd vector<cc2Xform> &paraPoses);
```

- `ccOCVMaxArrangement();`

  Default constructor.

- ```
  ccOCVMaxArrangement(
    const cmStd vector<ccOCVMaxParagraph> &paras,
    const cmStd vector<cc2Xform> &paraPoses);
  ```

  Constructs an arrangement using the given set of paragraphs.

  The paragraphs are arranged according to the affine poses.
  ```

Parameters

paras The set of paragraphs.

paraPoses The affine poses.

Throws

ccOCVMaxDefs::BadParams
The number of poses does not match the number of paragraphs.

- ```
  ccOCVMaxArrangement(const cmStd
    vector<ccOCVMaxParagraphPtrh_const> &paras,
    const cmStd vector<cc2Xform> &paraPoses);
  ```

  Const PtrHandle version of the constructor.

  Parameters

  paras The set of paragraphs.

  paraPoses The affine poses.

- ```
  ccOCVMaxArrangement(const cmStd
    vector<ccOCVMaxParagraphPtrh> &paras,
    const cmStd vector<cc2Xform> &paraPoses);
  ```

  PtrHandle version of the constructor.

  Parameters

  paras The set of paragraphs.

  paraPoses The affine poses.

  Notes

  The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

paragraphs

```
void paragraphs(const cmStd vector<ccOCVMaxParagraph>
 &paragraphs);

void paragraphs(const cmStd
 vector<ccOCVMaxParagraphPtrh_const> &paragraphs);

void paragraphs(const cmStd vector<ccOCVMaxParagraphPtrh>
 &paragraphs);
```

- ```
  void paragraphs(const cmStd vector<ccOCVMaxParagraph>
   &paragraphs);
  ```

  Sets the paragraphs of the arrangement.

  Parameters

  paragraphs The new paragraphs.

- ```
  void paragraphs(const cmStd
   vector<ccOCVMaxParagraphPtrh_const> &paragraphs);
  ```

  Const PtrHandle version of the setters.

  Parameters

  paragraphs The new paragraphs.

- ```
  void paragraphs(const cmStd vector<ccOCVMaxParagraphPtrh>
   &paragraphs);
  ```

  PtrHandle version of the setters.

  Parameters

  paragraphs The new paragraphs.

numParagraphs

```
const c_Int32 numParagraphs() const;
```

Returns the number of paragraphs in this arrangement.

paragraphs

```
const cmStd vector<ccOCVMaxParagraphPtrh_const>&
 paragraphs() const;
```

Gets the set of paragraphs for this arrangement.
```

Notes

Setter and getter are of different types.

paragraphPoses

```
const cmStd vector<cc2Xform>& paragraphPoses() const;

void paragraphPoses
  (const cmStd vector<cc2Xform> &paragraphPoses);
```

- `const cmStd vector<cc2Xform>& paragraphPoses() const;`

  Gets the paragraph poses for this arrangement with respect to the arrangement coordinate system.

- ```
  void paragraphPoses
    (const cmStd vector<cc2Xform> &paragraphPoses);
  ```

  Sets the paragraph poses for this arrangement with respect to the arrangement coordinate system.

  Parameters

  paragraphPoses The new paragraph poses.

origin

```
cc2Xform origin() const;

void origin(const cc2Xform& origin);
```

- `cc2Xform origin() const;`

  Gets the origin of the arrangement.

- `void origin(const cc2Xform& origin);`

  Sets the origin of the arrangement.

  Parameters

  origin The new origin.
```

encloseRect

```
ccRect encloseRect
  (const cc2Xform& clientFromArr = cc2Xform(),
  const cc2Xform& clientFromImage = cc2Xform()) const;
```

Returns the smallest rectangle aligned with the specified image coordinate system guaranteed to contain this arrangement given the transformation relating the arrangement coordinate frame plus a supplied offset to the client specified coordinate system, and the uncertainties in this transformation (translation, rotation, and scale), and the transformation relating the image coordinate system to the client coordinate system.

---

render

```
void render(ccPelBuffer<c_UInt8>& image,
  const cc2XformLinear& pose,
  c_UInt8 darkForegroundColor = 0,
  c_UInt8 lightForegroundColor = 255,
  c_UInt8 backgroundColor = 127,
  c_Int32 padding = 2) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cc2XformLinear& pose,
  const cmStd vector<cmStd vector<cmStd vector<c_Int32> > >&
    displayKeys,
  c_UInt8 darkForegroundColor = 0,
  c_UInt8 lightForegroundColor = 255,
  c_UInt8 backgroundColor = 127,
  c_Int32 padding = 2) const;

void render(ccPelBuffer<c_UInt8>& image,
  ccSynFontRenderOutline& outline,
  const cc2XformLinear& pose,
  const cmStd vector<cmStd vector<cmStd vector<c_Int32> > >&
    displayKeys = cmStd vector<cmStd vector<cmStd
    vector<c_Int32> > >(),
  c_UInt8 darkForegroundColor = 0,
  c_UInt8 lightForegroundColor = 255,
  c_UInt8 backgroundColor = 127,
  c_Int32 padding = 2,
  const ccColor& color = ccColor::greenColor()) const;

void render(ccSynFontRenderOutline& outline,
  const cc2XformBasePtrh_const& clientFromImageXform,
  const cc2XformLinear& pose,
  const cmStd vector<cmStd vector<cmStd vector<c_Int32> > >&
```

```
  displayKeys = cmStd vector<cmStd vector<cmStd
  vector<c_Int32> > >(),
const ccColor& color = ccColor::greenColor()) const;
```

- ```
  void render(ccPelBuffer<c_UInt8>& image,
   const cc2XformLinear& pose,
   c_UInt8 darkForegroundColor = 0,
   c_UInt8 lightForegroundColor = 255,
   c_UInt8 backgroundColor = 127,
   c_Int32 padding = 2) const;
  ```

  Renders the arrangement into an image with the given pose. Padding is the number of extra pixels to add on each side of the arrangement beyond the minimum required to render.

  Parameters

  image The image.

  pose The pose.

  darkForegroundColor
  The dark foreground color of the image.

  lightForegroundColor
  The light foreground color of the image.

  backgroundColor
  The background color of the image.

  padding The number of extra pixels to add on each side of the arrangement beyond the minimum required to render.

  Notes

  The *clientFromImage* xform of the image is used even if the image is unbound.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
   const cc2XformLinear& pose,
   const cmStd vector<cmStd vector<cmStd vector<c_Int32> > >&
    displayKeys,
   c_UInt8 darkForegroundColor = 0,
  ```

```
  c_UInt8 lightForegroundColor = 255,
  c_UInt8 backgroundColor = 127,
  c_Int32 padding = 2) const;
```

Renders the arrangement into an image with the given pose. Padding is the number of extra pixels to add on each side of the arrangement beyond the minimum required to render.

displayKeys specifies a character code to display in the rendered image instead of using the value of one of the arrangement's keys for that position. It should either be left completely empty or else it should have an identical structure to that of the arrangement (size of the outermost vector equals the number of paragraphs, size of the next vector equals the number of lines in that paragraph, size of the innermost vector equals the number of positions in that line).

Parameters

image The image.

pose The pose.

displayKeys The display keys.

darkForegroundColor
The dark foreground color of the image.

lightForegroundColor
The light foreground color of the image.

backgroundColor
The background color of the image.

padding The number of extra pixels to add on each side of the arrangement beyond the minimum required to render.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    ccSynFontRenderOutline& outline,
    const cc2XformLinear& pose,
    const cmStd vector<cmStd vector<cmStd vector<c_Int32> > >&
      displayKeys = cmStd vector<cmStd vector<cmStd
      vector<c_Int32> > >(),
    c_UInt8 darkForegroundColor = 0,
    c_UInt8 lightForegroundColor = 255,
  ```
  ```

```
c_UInt8 backgroundColor = 127,
c_Int32 padding = 2,
const ccColor& color = ccColor::greenColor()) const;
```

Renders the arrangement into an image with the given pose and an outline with the specified color. Padding is the number of extra pixels to add on each side of the arrangement beyond the minimum required to render.

displayKeys specifies a character code to display in the rendered image or outline instead of using the value of one of the arrangement's keys for that position. It should either be left completely empty or else it should have an identical structure to that of the arrangement (size of the outermost vector equals the number of paragraphs, size of the next vector equals the number of lines in that paragraph, size of the innermost vector equals the number of positions in that line).

Parameters

image The image.

outline The outline.

pose The pose.

displayKeys The display keys.

darkForegroundColor
The dark foreground color of the image.

lightForegroundColor
The light foreground color of the image.

backgroundColor
The background color of the image.

padding The number of extra pixels to add on each side of the arrangement beyond the minimum required to render.

color The color of the rendered arrangement in the image.

- ```
  void render(ccSynFontRenderOutline& outline,
    const cc2XformBasePtrh_const& clientFromImageXform,
    const cc2XformLinear& pose,
    const cmStd vector<cmStd vector<cmStd vector<c_Int32> > >&
  ```
  ```

```
   displayKeys = cmStd vector<cmStd vector<cmStd
   vector<c_Int32> > >(),
 const ccColor& color = ccColor::greenColor()) const;
```

Renders the arrangement into an outline with the given pose and the specified color.

displayKeys specifies a character code to display in the rendered outline instead of using the value of one of the arrangement's keys for that position. It should either be left completely empty or else it should have an identical structure to that of the arrangement (size of the outermost vector equals the number of paragraphs, size of the next vector equals the number of lines in that paragraph, size of the innermost vector equals the number of positions in that line).

Parameters

outline The outline.

clientFromImageXform
A **cc2Xform** describing the transformation between image coordinates and client coordinates.

pose The pose.

displayKeys The display keys.

color The color of the rendered arrangement in the image.

clone

```
ccOCVMaxArrangementPtrh clone() const;
```

Returns a deep copy of this arrangement, where each paragraph is a clone of the corresponding paragraph of this arrangement.

operator==

```
bool operator==(const ccOCVMaxArrangement& other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxArrangement& other) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

Typedefs

ccOCVMaxArrangementPtrh

```
typedef ccPtrHandle<ccOCVMaxArrangement>
  ccOCVMaxArrangementPtrh;
```

ccOCVMaxArrangementPtrh_const

```
typedef ccPtrHandle_const<ccOCVMaxArrangement>
  ccOCVMaxArrangementPtrh_const;
```

ccOCVMaxArrangementSearchKeySets

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxArrangementSearchKeySets;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class is used to specify what key sets should be searched for at runtime. It requires that the structure exactly matches the structure of the trained arrangement being searched for. It also requires that the specified key sets be subsets of the trained arrangement's keysets. If an individual search key set is empty, the runtime search key set will be the full key set that was trained.

Constructors/Destructors

ccOCVMaxArrangementSearchKeySets

```
ccOCVMaxArrangementSearchKeySets();

ccOCVMaxArrangementSearchKeySets(const ccOCVMaxArrangement
  &arr);
```

- `ccOCVMaxArrangementSearchKeySets();`

- `ccOCVMaxArrangementSearchKeySets(const ccOCVMaxArrangement &arr);`

  Construct an arrangement-keysets object from an arrangement. The structure of the constructed object hierarchy will match the arrangement structure, and all key sets will be empty.

  Notes

  The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

paragraphKeySetsVect

```
const cmStd vector<ccOCVMaxParagraphSearchKeySets>
 &paragraphKeySetsVect() const;

void paragraphKeySetsVect(const cmStd
 vector<ccOCVMaxParagraphSearchKeySets>
 &paragraphKeySets);
```

- ```
  const cmStd vector<ccOCVMaxParagraphSearchKeySets>
   &paragraphKeySetsVect() const;
  ```

  Gets the entire vector of paragraph-keysets objects.

- ```
  void paragraphKeySetsVect(const cmStd
   vector<ccOCVMaxParagraphSearchKeySets>
   &paragraphKeySets);
  ```

  Sets the entire vector of paragraph-keysets objects.

  Parameters

  paragraphKeySets
  The paragraph-keysets objects.

paragraphKeySets

```
const ccOCVMaxParagraphSearchKeySets
 &paragraphKeySets(c_Int32 index) const;

void paragraphKeySets(c_Int32 index, const
 ccOCVMaxParagraphSearchKeySets &keys);
```

- ```
  const ccOCVMaxParagraphSearchKeySets
   &paragraphKeySets(c_Int32 index) const;
  ```

  Gets a single paragraph-keysets object.

  Parameters

  index The index of the paragraph-keysets object.

  Throws

  ccOCVMaxDefs::BadParams
  index is out of range.
  ```

- ```
  void paragraphKeySets(c_Int32 index, const
    ccOCVMaxParagraphSearchKeySets &keys);
  ```

  Sets a single paragraph-keysets object.

  Parameters

  index The index of the paragraph-keysets object.

  Throws

  ccOCVMaxDefs::BadParams
  index is out of range.

positionKeySets

```
const ccOCVMaxKeySet &positionKeySets(c_Int32
  paragraphIndex, c_Int32 lineIndex, c_Int32 positionIndex)
  const;

void positionKeySets(c_Int32 paragraphIndex, c_Int32
  lineIndex, c_Int32 positionIndex, const ccOCVMaxKeySet
  &keys);
```

- ```
  const ccOCVMaxKeySet &positionKeySets(c_Int32
    paragraphIndex, c_Int32 lineIndex, c_Int32 positionIndex)
    const;
  ```

  Gets a single positional keyset object. This function is a convenient way to "drill down" and get a single value deep in a hierarchy.

  Parameters

  paragraphIndex The paragraph index.

  lineIndex The line index.

  positionIndex The position index.

  Throws

  ccOCVMaxDefs::BadParams
  Any of the indices are out of range.

- `void positionKeySets(c_Int32 paragraphIndex, c_Int32 lineIndex, c_Int32 positionIndex, const ccOCVMaxKeySet &keys);`

  Sets a single positional keyset object. This function is a convenient way to "drill down" and set a single value deep in a hierarchy.

  Parameters

  paragraphIndex The paragraph index.

  lineIndex The line index.

  positionIndex The position index.

  Notes

  The setter requires that the all indices are valid. This function cannot be used to add new elements to a SearchKeySets object. For example, you cannot set position 6 on a line with only 3 position elements.

  Throws

  ccOCVMaxDefs::BadParams
  Any of the indices are out of range.

appendParagraphKeySets

```
void appendParagraphKeySets(const
 ccOCVMaxParagraphSearchKeySets &paraKeySets);
```

Adds a single paragraph-keyset object to the end of the arrangement.

Parameters

paraKeySets The paragraph-keyset object to be added.

numParagraphKeySets

```
c_Int32 numParagraphKeySets() const;
```

Returns the number of paragraph keySets stored.

Operators

operator==

```
bool operator==(const ccOCVMaxArrangementSearchKeySets&
 other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxArrangementSearchKeySets&
  other) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for inequality.

ccOCVMaxArrangementSearchKeySets

ccOCVMaxDefs

```
#include <ch_cvl\ocmax.h>

class ccOCVMaxDefs;
```

A name space that holds enumerations used with the OCVMax tool.

Enumerations

PositionStatus

```
enum PositionStatus;
```

This enumeration defines verification status results for positions.

Value	Meaning
eVerified	The character passes verification. The matching score is greater than the accept threshold, and the difference between the matching score and the highest scoring confusing character is greater than the confidence threshold.
eConfused	The matching score for the character is greater than the accept threshold, however the difference between the matching score and the highest scoring confusing character is not greater than the confidence threshold.
eFailed	The character fails verification. Its matching score is less than the accept threshold.

DrawFlags

```
enum DrawFlags;
```

This enumeration determines which graphics are included when diagnostics are generated.

Value	Meaning
eDrawArrangement = 0x001	Draws the final arrangement pose.
eDrawParagraphs = 0x002	Draws the final paragraph poses.
eDrawPositions = 0x004	Draws the final character poses.
eDrawArrangementBox = 0x008	Draws the bounding box with the arrangement pose.

Value	Meaning
eDrawParagraphBoxes = 0x010	Draws the bounding box with the paragraph pose.
eDrawPositionBoxes = 0x020	Draws the bounding box with the character pose.
eDrawArrangementOutlines = 0x040	Draws character outlines with the arrangement pose.
eDrawParagraphOutlines = 0x080	Draws character outlines with the paragraph pose.
eDrawPositionOutlines = 0x100	Draws character outlines with the character pose.
eDrawArrangementText = 0x200	Draws character text with the arrangement pose.
eDrawParagraphText = 0x400	Draws character text with the paragraph pose.
eDrawPositionText = 0x800	Draws character text with the character pose.

Polarity

```
enum Polarity;
```

Polarity types for characters to specify the appearance of the character relative to its nearby background in the image.

Value	Meaning
eDarkOnLight = 0	Dark characters on light background
eLightOnDark	Light characters on dark background
eUnknown	Unknown polarity

VerificationType

```
enum VerificationType;
```

Verification type to indicate how to perform verification at a character position.

Value	Meaning
eNormal = 0	Uses standard verification.
eAlwaysVerified	Uses standard verification, but then always marks result as verified.
eIgnore	Performs no verification, but marks result as verified and having a score of 1.0.

DOF

```
enum DOF;
```

Degrees of freedom. Refer to the descriptions of the named degrees of freedom of a **cc2Matrix** for each OCVMax degree of freedom.

Value	Meaning
eAngle = 0x01	Rotation angle (**cc2Matrix::rotation()**)
eUniformScale = 0x02	Uniform scale (**cc2Matrix::scale()**)
eXScale = 0x04	xScale (**cc2Matrix::xScale()**)
eYScale = 0x08	yScale (**cc2Matrix::yScale()**)
eShear = 0x10	Shear (**cc2Matrix::shear()**)
eAspect = 0x20	Aspect (**cc2Matrix::aspect()**) Used only in reporting zone violations.
eXTranslation = 0x4000	x-translation Used only in reporting zone violations.
eYTranslation = 0x8000	y-translation Used only in reporting zone violations.

CharacterRegistration

```
enum CharacterRegistration;
```

Character registration type to specify matching technology.

Value	Meaning
eStandard = 0x01	Uses PatMax for registration.
eCorrelation = 0x08	Uses correlation for registration.

ScoreMode

```
enum ScoreMode;
```

Score mode used to compute character score when using standard (PatMax) character registration.

Value	Meaning
eScoreUsingClutter = 1	Reduces the character score if extra features (clutter) are present.
eScoreWithoutClutter = 2	Computes the score based only on the presence of expected features, ignoring any extra features (clutter).
eScoreUsingOptimizedClutter = 3	Reduces the character score if extra features (clutter) are present, but optimizes score for relatively low-contrast clutter.

PoseCompute `enum PoseCompute;`

Specifies which result poses to compute.

Value	Meaning
ePoseComputeLineLinear = 0x01	See **ccOCVMaxRunParams::computePoses()** on page 2270.
ePoseComputeParagraphLinear = 0x02	
ePoseComputeArrangementLinear = 0x04	
ePoseComputeArrangementNonlinear = 0x08	

ccOCVMaxDefs

ccOCVMaxKeySet

```
#include <ch_cvl\ocmax.h>

class ccOCVMaxKeySet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The **ccOCVMaxKeySet** class contains information about the font key values that are relevant to a particular position within an **ccOCVMaxLine** object. It contains the following information:

Keys:

The set of unique keys for characters that may be used at the position. For example, if only numbers are possible for the position, this set would contain all of the keys for numbers and none for letters. An empty keyset indicates that all keys in the owning paragraphs *alphabetKeys* vector should be considered.

Type:

The **ccOCVMaxDefs::VerificationType** value indicates how to apply optical character verification (OCV) using the keyset. *eNormal* is standard OCV, which will return values of *eVerified*, *eConfused*, or *eFailed*. *eAlwaysVerified* is also standard OCV and will score every key in the keyset at the appropriate position, but it will always return *eVerified* along with the score and ID of the highest-scoring key. *eIgnore* will not do any processing at all at the appropriate position, and will not return an ID, but will always return *eVerified* and a score of 1.

Constructors/Destructors

ccOCVMaxKeySet

```
ccOCVMaxKeySet(
  const cmStd vector<c_Int32>& theKeys =
    cmStd vector<c_Int32>(),
  ccOCVMaxDefs::VerificationType type =
    ccOCVMaxDefs::eNormal);
```

Constructs a key set using the supplied keys and verification type.

Parameters

theKeys The set of unique keys for characters that can be found at this position. For example, if only numbers are possible for this position in the paragraph, *theKeys* contains all the of the keys for numbers and none for letters.

If *theKeys* is empty, all alphabet keys for the current font are considered.

type Indicates how to apply OCV using the key set. *eNormal* is standard for OCV and returns an indication of verified, confused or failed at each position in the paragraph.

Throws

ccOCVMaxDefs::BadParams
Any of the supplied keys are the same.

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

keys

```
const cmStd vector<c_Int32>& keys() const;
void keys(const cmStd vector<c_Int32>& ks);
```

- `const cmStd vector<c_Int32>& keys() const;`

Gets the vector of valid characters for this position within the paragraph.

- `void keys(const cmStd vector<c_Int32>& ks);`

Sets the vector of characters that will be considered at this particular position within the paragraph.

Parameters

ks The vector of key values, empty by default.

Throws

ccOCVMaxDefs::BadParams
More than one element of the vector has the same value.

verificationType

```
ccOCVMaxDefs::VerificationType verificationType() const;

void verificationType(
 ccOCVMaxDefs::VerificationType type);
```

- `ccOCVMaxDefs::VerificationType verificationType() const;`

  Gets the verification type performed at this character location within the paragraph.

- `void verificationType(`
  `ccOCVMaxDefs::VerificationType type);`

  Sets the verification type to perform at this character location:

Type	Description
eNormal	Default value. Returns *ccOCVMaxDefs::eFailed* if no element of the key set had a score above the accept threshold. Returns *ccOCVMaxDefs::eVerified* if an element of the key set had a score above the accept threshold and its confidence score (the difference between its score and that of its highest-scoring confusible character) is above the confidence threshold. Returns *ccOCVMaxDefs::eConfused* otherwise.
eAlwaysVerified	Returns *ccOCVMaxDefs::eVerified* regardless of the scores along with the score and ID of the highest-scoring key.
eIgnore	Returns *ccOCVMaxDefs::eVerified* regardless of the scores and a score of 1.

Parameters

type The type of verification performed at this character location.

Operators

operator==

```
bool operator==(const ccOCVMaxKeySet& other) const;
```

Returns true if and only if this key set is equal to the specified other key set.

Parameters

other The pointer to the other key set.

operator!= `bool operator!=(const ccOCVMaxKeySet& other) const;`

Returns true if this key set is not equal to the other key set and false otherwise.

Parameters

other The pointer to the other key set.

ccOCVMaxLine

```
#include <ch_cvl\ocmax.h>

class ccOCVMaxLine : public virtual ccPersistent,
                     public virtual ccRepBase
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The OCVMax Line class represents a sequential, spatial arrangement of characters chosen from a single font. In general, the character arrangement is such that each character has spatial neighbors that are the previous and next characters in the sequence.

The spatial arrangement of characters is specified with respect to a **ccOCVMaxLine** coordinate frame. For each of the specified character positions in the line, a set of possible character keys can be supplied. The more character keys per position, the more characters a particular OC vision tool (for example, OCVMax) will consider. Specifying more than one key for a position will in general require more memory for such a tool, but will allow on-the-fly replacement of characters in the line without substantial re-training.

A typical **ccOCVMaxLine** is a horizontal line of upright characters (which can easily be specified using the first constructor).

The primary difference between a **ccOCVMaxLine** and a **ccOCLine** is that a **ccOCVMaxLine** is built with a font rather than an alphabet.

Constructors/Destructors

ccOCVMaxLine `ccOCVMaxLine();`

The default constructor.

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

keySetSequence

```
void keySetSequence(const cmStd vector<ccOCVMaxKeySet>
 &keySetSequence);

void keySetSequence(const cmStd vector<ccOCVMaxKeySet>
 &keySetSequence,
 const cmStd vector<c_Int32> &displayIndices);

const cmStd vector<ccOCVMaxKeySet> &keySetSequence() const;
```

- ```
  void keySetSequence(const cmStd vector<ccOCVMaxKeySet>
   &keySetSequence);
  ```

  Sets the line to use the specified sequence of character key sets.

  Parameters

  keySetSequence Sequence of character key sets.

- ```
  void keySetSequence(const cmStd vector<ccOCVMaxKeySet>
   &keySetSequence,
   const cmStd vector<c_Int32> &displayIndices);
  ```

  Sets the line to use the specified sequence of character key sets. *displayIndices* is a vector that indicates which element of each keyset should be used if the line is rendered for display purposes. The element is specified by an index, so a value of 0 will render the first element of each keyset. An index of -1 indicates that nothing should be rendered in that slot (that is, blank).

  Parameters

  keySetSequence Sequence of character key sets.

  displayIndices Element indices.

  Notes

  The one-argument version of this function assumes a vector of *displayIndices* filled with the value 0.

  If a keyset on the line has **isWildcard()** true, then its values will be rendered with the owning paragraph's current **wildcardDisplayKey()**.

  Throws

  ccOCVMaxDefs::BadParams
  The lengths of the two vectors are different.

BadIndex

The *displayIndices* vector contains any indices that are outside the range of keys in the respective **ccOCVMaxKeySet** in *keySetSequence*, and not -1.

- `const cmStd vector<ccOCVMaxKeySet> &keySetSequence() const;`

  Gets the *keySetSequence* vector.

displayIndices

```
const cmStd vector<c_Int32> &displayIndices() const;
```

Gets the *displayIndices* vectors.

numPositions `c_Int32 numPositions() const;`

Returns the number of character positions in this line.

operator== `bool operator==(const ccOCVMaxLine& other) const;`

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!= `bool operator!=(const ccOCVMaxLine& other) const;`

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

Typedefs

ccOCVMaxLinePtrh

```
typedef ccPtrHandle<ccOCVMaxLine> ccOCVMaxLinePtrh;
```

ccOCVMaxLinePtrh_const

```
typedef ccPtrHandle_const<ccOCVMaxLine>
 ccOCVMaxLinePtrh_const;
```

ccOCVMaxLine

ccOCVMaxLineResult

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxLineResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxLineResult** class contains all the result information for a single OCVMax Line.

Constructors/Destructors

ccOCVMaxLineResult

```
ccOCVMaxLineResult();
```

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

foundPose

```
bool foundPose() const;
```

Returns whether a pose was determined for this result.

clientPose

```
const cc2Xform& clientPose() const;
```

Returns the pose of the line in client coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundPose()** operation returned false.

foundParagraphPose

```
bool foundParagraphPose() const;
```

Returns whether a paragraph pose was determined for this result.

paragraphPose

```
const cc2Xform& paragraphPose() const;
```

Returns the pose of the line in paragraph coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundParagraphPose()** operation returned false.

foundArrangementPose

```
bool foundArrangementPose() const;
```

Returns whether an arrangement pose was determined for this result.

arrangementPose

```
const cc2Xform& arrangementPose() const;
```

Returns the pose of the line in arrangement coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundArrangementPose()** operation returned false.

score

```
double score() const;
```

Returns the overall score of the line, which is the average of all of the line's position scores which have VerificationType, eNormal, or eAlwaysVerified.

numPositions

```
c_Int32 numPositions() const;
```

Returns the total number of character positions.

numPositionsVerified

```
c_Int32 numPositionsVerified() const;
```

Returns the number of character positions correctly verified.

numPositionsEmpty

```
c_Int32 numPositionsEmpty() const;
```

Returns the number of character positions that are empty.

verified

```
bool verified() const;
```

Returns true if the entire line was verified (all character positions have been verified), and false otherwise.

keyZoneViolations

```
c_UInt32 keyZoneViolations() const;
```

Returns which zones of the key search parameters were violated by at least one position and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

fullZoneViolations

```
c_UInt32 fullZoneViolations() const;
```

Return which zones of the full (that is, image or start pose) search parameters were violated by at least one position and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

positionResults

```
const cmStd vector<ccOCVMaxPositionResult>&
  positionResults() const;
```

Returns the results for all the character positions for which verification has been attempted.

Operators

operator==

```
bool operator==(const ccOCVMaxLineResult& other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxLineResult& other) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for inequality.

ccOCVMaxLineResult

ccOCVMaxLineSearchKeySets

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxLineSearchKeySets;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class is used to specify what key sets should be searched for at runtime. It requires that the structure exactly matches the structure of the trained arrangement being searched for. It also requires that the specified key sets be subsets of the trained arrangement's keysets. If an individual search key set is empty, the runtime search key set will be the full key set that was trained.

Public Member Functions

keySetSequence

```
const cmStd vector<ccOCVMaxKeySet> &keySetSequence() const;

void keySetSequence(const cmStd vector<ccOCVMaxKeySet>
  &posKeySets);
```

- `const cmStd vector<ccOCVMaxKeySet> &keySetSequence() const;`

  Gets the entire vector of keyset objects for the line.

- `void keySetSequence(const cmStd vector<ccOCVMaxKeySet> &posKeySets);`

  Sets the entire vector of keyset objects for the line.

  Notes

  The default constructor, destructor, copy constructor, and assignment operator are used.

keySet

```
const ccOCVMaxKeySet &keySet(c_Int32 index) const;
void keySet(c_Int32 index, const ccOCVMaxKeySet &keyset);
```

- `const ccOCVMaxKeySet &keySet(c_Int32 index) const;`

  Gets a single keyset object.

  Parameters
  index The keyset object index.

  Throws
  ccOCVMaxDefs::BadParams
  index is out of range.

- `void keySet(c_Int32 index, const ccOCVMaxKeySet &keyset);`

  Sets a single keyset object.

  Parameters
  index The keyset object index.

  Throws
  ccOCVMaxDefs::BadParams
  index is out of range.

appendKeySet

```
void appendKeySet(const ccOCVMaxKeySet &keyset);
```

Add a single keyset object to the end of the line.

numKeySets

```
c_Int32 numKeySets() const;
```

Returns the number of key sets stored.

Operators

operator==

```
bool operator==(const ccOCVMaxLineSearchKeySets& other)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters
other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxLineSearchKeySets& other)
  const;
```

Return true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxLineSearchKeySets

ccOCVMaxParagraph

```
#include <ch_cvl\ocmax.h>

class ccOCVMaxParagraph : public virtual ccPersistent,
                          public virtual ccRepBase
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The OCVMax Paragraph class represents a sequential, spatial arrangement of lines chosen from a single font. The line arrangement is such that each line is directly below the previous line in the paragraph.

Constructors/Destructors

ccOCVMaxParagraph

```
ccOCVMaxParagraph();
```

Constructs a paragraph.

The lines are arranged with a uniform spacing (with respect to their origins) at position intervals given by the line spacing parameters (extraLeading, combined with the font's leading value) in the order they are specified.

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

lines

```
void lines(const cmStd vector<ccOCVMaxLine> &lines);

void lines(const ccOCVMaxLine &line);
```

- ```
  void lines(const cmStd vector<ccOCVMaxLine> &lines);
  ```

  Sets the lines of the paragraph.

  Parameters

lines	The new lines.
  ```

- `void lines(const ccOCVMaxLine &line);`

  Sets the a line of the paragraph.

  Parameters
  line The new line.

numLines `c_Int32 numLines() const;`

Returns the number of lines in this paragraph.

line `const ccOCVMaxLine &line(c_Int32 posIndex) const;`

Returns the line at the specified position in the paragraph.

Parameters
posIndex The position index.

Throws
ccOCVMaxDefs::BadIndex
The specified line index is invalid.

lines `const cmStd vector<ccOCVMaxLine>& lines() const;`

Returns a vector of all the lines.

---

font `const ccSynFontPtrh_const& font() const;`

`void font(const ccSynFontPtrh_const&);`

- `const ccSynFontPtrh_const& font() const;`

  Gets the font for this paragraph.

- `void font(const ccSynFontPtrh_const&);`

  Sets the font for this paragraph.

  Parameters
  ccSynFontPtrh_const

alphabetKeys

```
const cmStd vector<c_Int32> &alphabetKeys() const;

void alphabetKeys(const cmStd vector<c_Int32>
 &alphabetKeys);
```

- `const cmStd vector<c_Int32> &alphabetKeys() const;`

  Gets the list of font characters that will be used to build the OCV alphabet for this paragraph.

- `void alphabetKeys(const cmStd vector<c_Int32> &alphabetKeys);`

  Sets the list of font characters that will be used to build the OCV alphabet for this paragraph.

  Only chars in the alphabet are considered for confusion. If *alphabetChars* is empty, all characters specified in the font will be used to build the alphabet.

  Parameters

  alphabetKeys The list of font characters.

origin

```
cc2Xform origin() const;

void origin(const cc2Xform& origin);
```

- `cc2Xform origin() const;`

  Gets the origin of the paragraph.

- `void origin(const cc2Xform& origin);`

  Sets the origin of the paragraph.

  Parameters

  origin The origin.

tracking

```
cc2Vect tracking() const;

void tracking(const cc2Vect &tracking);
```

- `cc2Vect tracking() const;`

  Gets the tracking for this paragraph with respect to the previous char in each line, in the font's coordinate system.

- `void tracking(const cc2Vect &tracking);`

  Sets the tracking for this paragraph with respect to the previous char in each line, in the font's coordinate system.

  Tracking is an amount added to the default advance between characters.

  Parameters
  tracking The new tracking.

extraLeading

```
cc2Vect extraLeading() const;
void extraLeading(const cc2Vect &extraLeading);
```

- `cc2Vect extraLeading() const;`

  Gets the extra leading for this paragraph, in the font's coordinate system.

- `void extraLeading(const cc2Vect &extraLeading);`

  Sets the extra leading for this paragraph, in the font's coordinate system.

  The origin-to-origin displacement between successive lines is equal to the sum of the font's leading [as a vector (0, leading)] and this value.

  Parameters
  extraLeading The new extra leading.

  Notes
  "Leading" is pronounced "ledding".

polarity

```
ccOCVMaxDefs::Polarity polarity() const;
void polarity(ccOCVMaxDefs::Polarity polarity);
```

- `ccOCVMaxDefs::Polarity polarity() const;`

  Gets the polarity value.

- `void polarity(ccOCVMaxDefs::Polarity polarity);`

  Sets the polarity value.

  Parameters
  polarity The new polarity value.

confusionThreshold

```
double confusionThreshold() const;

void confusionThreshold(double thresh);
```

- `double confusionThreshold() const;`

  Gets the confusion threshold for the alphabet.

  Throws
  ccOCVMaxDefs::BadParams
  The threshold is negative or greater than 1.0.

- `void confusionThreshold(double thresh);`

  Sets the confusion threshold for the alphabet.

  Parameters
  thresh The new confusion threshold value.

spotSizeFactor

```
double spotSizeFactor() const;

void spotSizeFactor(double spotSizeFactor);
```

- `double spotSizeFactor() const;`

  Gets the spot size factor, which adjusts font rendering for the paragraph by multiplying the size of the "beam" used to scribe the characters.

- `void spotSizeFactor(double spotSizeFactor);`

  Sets the spot size factor, which adjusts font rendering for the paragraph by multiplying the size of the "beam" used to scribe the characters.

  Parameters
  spotSizeFactor The new spot size factor.

  Throws
  ccOCVMaxDefs::BadParams
  spotSizeFactor is less than or equal to 0.

extraStrokeWidth

```
double extraStrokeWidth() const;

void extraStrokeWidth(double extraStrokeWidth);
```

- `double extraStrokeWidth() const;`

  Gets the *extraStrokeWidth*, which adjusts font rendering for the paragraph using morphology.

- `void extraStrokeWidth(double extraStrokeWidth);`

  Sets the *extraStrokeWidth*, which adjusts font rendering for the paragraph using morphology.

  A negative value indicates that the stroke width should be decreased.

  Parameters

  extraStrokeWidth The new *extraStrokeWidth* value.

spotSpacingXScale

```
double spotSpacingXScale() const;

void spotSpacingXScale(double xScale);
```

- `double spotSpacingXScale() const;`

  Gets the factor by which to change the x-coordinate of spots relative to each other; also changes the endpoints of strokes.

- `void spotSpacingXScale(double xScale);`

  Sets the factor by which to change the x-coordinate of spots relative to each other; also changes the endpoints of strokes.

  Parameters

  xScale The new factor value.

  Throws

  ccOCVMaxDefs::BadParams

  xScale is less than or equal to 0.

spotSpacingYScale

```
double spotSpacingYScale() const;

void spotSpacingYScale(double yScale);
```

- `double spotSpacingYScale() const;`

  Gets the factor by which to change the y-coordinate of spots relative to each other; also changes the endpoints of strokes.

- `void spotSpacingYScale(double yScale);`

  Sets the factor by which to change the y-coordinate of spots relative to each other; also changes the endpoints of strokes.

  Parameters

  yScale The new factor value.

  Throws

  ccOCVMaxDefs::BadParams

  yScale is less than or equal to 0.

encloseRect

```
ccRect encloseRect
  (const cc2Xform& clientFromParagraph = cc2Xform(),
  const cc2Xform& clientFromImage = cc2Xform()) const;
```

Returns the smallest rectangle aligned with the specified image coordinate system guaranteed to contain this paragraph given the transformation relating the paragraph coordinate frame and the transformation relating the image coordinate system to the client coordinate system.

markRectChar

```
ccAffineRectangle markRectChar(c_Int32 lineIndex,
  c_Int32 posIndex,
  c_Int32 keyIndex = 0) const;
```

Returns the mark rect for the given position, in the font's coordinate system.

Notes

The mark rect depends only on the character keys, not on the position within a line.

The mark rect for a blank is degenerate.

The *keyIndex* is the index of the particular key in the keyset at the given position.

Parameters

lineIndex The line index.

posIndex The position index.

keyIndex The key index.

Throws

ccOCVMaxDefs::BadParams
Any index is negative or larger than the number of lines/positions/keys present in the paragraph.

encloseRectChar

```
ccRect encloseRectChar(c_Int32 lineIndex,
 c_Int32 posIndex,
 const cc2Xform& clientFromParagraph = cc2Xform(),
 const cc2Xform& clientFromImage = cc2Xform()) const;
```

Returns the smallest rectangle aligned with the specified image coordinate system guaranteed to contain the specified char given the transform relating the paragraph coordinate frame and the transformation relating the image coordinate system to the client coordinate system.

Parameters

lineIndex The line index.

posIndex The position index.

Throws

ccOCVMaxDefs::BadParams
lineIndex is greater than or equal to *numLines, or* *posIndex* is greater than or equal to lines[*lineIndex*].*numPositions*

nextLine

```
cc2Vect nextLine() const;
```

Returns the location of the origin of the next line of text beyond those specified in the paragraph. For example, if the paragraph has three lines, this is where the fourth line would start.

nextChar

```
cc2Vect nextChar() const;
```

Returns the location of the origin of the next character of text beyond those specified in the paragraph, based on the longest line in the paragraph. However, the x coordinate is set as if the character was part of the paragraph's first line. For example, if the paragraph consists of the lines "123" and "12345", the given location would be six characters out from the first line (the star in this diagram):

123..*

12345

Motivation: this is where another adjacent paragraph might start.

wildcardDisplayKey

```
void wildcardDisplayKey(c_Int32 wcDisplayKey);

c_Int32 wildcardDisplayKey() const;
```

- `void wildcardDisplayKey(c_Int32 wcDisplayKey);`

  Sets the font key value that is used to display wildcards (that is, keysets consisting of an empty list) when the paragraph's **render()** method is called.

  Parameters

  wcDisplayKey The new font key value.

- `c_Int32 wildcardDisplayKey() const;`

  Gets the font key value that is used to display wildcards (that is, keysets consisting of an empty list) when the paragraph's **render()** method is called.

clone

```
ccOCVMaxParagraphPtrh clone() const;
```

Returns a copy of this paragraph.

fontAvailableChars

```
const cmStd vector<c_Int32>& fontAvailableChars() const;
```

Gets the available characters in the font.

Notes

This differs from **font()**->**availableChars(false)** if the keys in the keys sets contain blanks with different keys.

operator== `bool operator==(const ccOCVMaxParagraph& other) const;`

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

Notes

The fonts in the two lines are considered equal if and only if they are *ptrHandles* to the same underlying object.

operator!= `bool operator!=(const ccOCVMaxParagraph& other) const;`

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

Typedefs

ccOCVMaxParagraphPtrh

```
typedef ccPtrHandle<ccOCVMaxParagraph>
  ccOCVMaxParagraphPtrh;
```

ccOCVMaxParagraphPtrh_const

```
typedef ccPtrHandle_const<ccOCVMaxParagraph>
  ccOCVMaxParagraphPtrh_const;
```

ccOCVMaxParagraphResult

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxParagraphResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxParagraphResult** class contains result information for a single paragraph.

Constructors/Destructors

ccOCVMaxParagraphResult

```
ccOCVMaxParagraphResult();
```

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

foundPose

```
bool foundPose() const;
```

Returns whether a pose was determined for this result.

clientPose

```
const cc2Xform& clientPose() const;
```

Returns the pose of the paragraph in client coordinates.

Throws

ccOCVMaxDefs::NotFound

The **foundPose()** operation returned false.

foundArrangementPose

```
bool foundArrangementPose() const;
```

Returns whether an arrangement pose was determined for this result.

arrangementPose

```
const cc2Xform& arrangementPose() const;
```

Returns the pose of the paragraph in arrangement coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundArrangementPose()** operation returned false.

score

```
double score() const;
```

Returns the overall score of the paragraph (the average of the scores of all of the characters in the paragraph).

numPositions

```
c_Int32 numPositions() const;
```

Returns the total number of character positions (on all lines).

numPositionsVerified

```
c_Int32 numPositionsVerified() const;
```

Returns the number of character positions (on all lines) correctly verified.

numPositionsEmpty

```
c_Int32 numPositionsEmpty() const;
```

Returns the number of character positions that are empty.

numLines

```
c_Int32 numLines() const;
```

Returns the total number of lines.

numLinesVerified

```
c_Int32 numLinesVerified() const;
```

Returns the number of lines correctly verified.

verified

```
bool verified() const;
```

Returns true if the entire paragraph was verified (all character positions have been verified), and false otherwise.

keyZoneViolations

```
c_UInt32 keyZoneViolations() const;
```

Returns which zones of the key search parameters were violated by at least one position and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

fullZoneViolations

```
c_UInt32 fullZoneViolations() const;
```

Returns which zones of the full (that is, image or start pose) search parameters were violated by at least one position and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

lineResults

```
const cmStd vector<ccOCVMaxLineResult>& lineResults()
  const;
```

Returns the results for all the lines in the paragraph.

Operators

operator==

```
bool operator==(const ccOCVMaxParagraphResult& other)
  const;
```

Returns true if this object is equal to to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxParagraphResult& other)
const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxParagraphResult

ccOCVMaxParagraphRunParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxParagraphRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The OCVMax paragraph runtime parameters class encapsulates all of the parameters for a particular paragraph within an arrangement that are required for character verification, including:

confusionThreshold

Each paragraph builds a "confusion matrix" when it's trained. This matrix has a number for every pair of keys in the paragraph's font alphabet. The number indicates how confusable the two keys are, with values near one being very confusable (e.g. '5' and 'S') and values near zero being not confusable (e.g. '5' and '1'). The *confusionThreshold* parameter specifies the value below which other keys are not considered by the tool for confusion purposes. For example, if the tool is expecting to find a '5', it will also check other potentially confusing keys to see if they are also good matches. With a confusion threshold of 0.9, the keys for 'S' and '6' might be tried, but '1' would not. With a threshold of 0.4, many somewhat-similar keys would be tried, such as 'E' or '8'. With a threshold of 0, all keys would be tried.

confidenceThreshold

It specifies the amount that the score of the highest-scoring character in the current key set must be above the highest-scoring character that has a non-zero confusion score with any character in the key set.
Note that if a potentially confusing character has a higher score than the highest-scoring key set member, then the status result will always be "confused", but the returned key will still be the one from the key set, not the confusing key.
Note that only keys that pass the *confusionThreshold* test, above, are considered for this scoring/verification step.

Constructors/Destructors

ccOCVMaxParagraphRunParams

```
ccOCVMaxParagraphRunParams(double confusionThreshold =
  0.5, double confidenceThreshold = 0.0);
```

Constructs a runtime parameters object with the supplied parameters.

Parameters

confusionThreshold The confusion threshold.

confidenceThreshold The confidence threshold.

Throws

ccOCVMaxDefs::BadParams
The confusion threshold value is less than 0 or greater than 1 or the confidence threshold value is less than 0 or greater than 1.

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

confidenceThreshold

```
double confidenceThreshold() const;

void confidenceThreshold(double t);
```

- `double confidenceThreshold() const;`

Gets the confidence threshold.

Throws

ccOCVMaxDefs::BadParams
t is less than 0 or greater than 1.

- `void confidenceThreshold(double t);`

Sets the confidence threshold.

Parameters

t The new confidence threshold value.

confusionThreshold

```
double confusionThreshold() const;

void confusionThreshold(double t);
```

- `double confusionThreshold() const;`

  Gets the confusion threshold

  Throws

  ccOCVMaxDefs::BadParams
  : *t* is less than 0 or greater than 1.

- `void confusionThreshold(double t);`

  Sets the confusion threshold.

  Parameters

t	The new confusion threshold value.

Operators

operator==

```
bool operator==(const ccOCVMaxParagraphRunParams& other)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxParagraphRunParams& other)
  const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for inequality.

ccOCVMaxParagraphRunParams

ccOCVMaxParagraphSearchKeySets

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxParagraphSearchKeySets;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class is used to specify what key sets should be searched for at runtime. It requires that the structure exactly matches the structure of the trained arrangement being searched for. It also requires that the specified key sets be subsets of the trained arrangement's keysets. If an individual search key set is empty, the runtime search key set will be the full key set that was trained.

Public Member Functions

lineKeySetsVect

```
const cmStd vector<ccOCVMaxLineSearchKeySets>
  &lineKeySetsVect() const;

void lineKeySetsVect(const cmStd
  vector<ccOCVMaxLineSearchKeySets> &lineKeySets);
```

- ```
  const cmStd vector<ccOCVMaxLineSearchKeySets>
    &lineKeySetsVect() const;
  ```

  Gets the entire vector of line-keysets objects.

- ```
  void lineKeySetsVect(const cmStd
    vector<ccOCVMaxLineSearchKeySets> &lineKeySets);
  ```

  Sets the entire vector of line-keysets objects.

  Notes

  The default constructor, destructor, copy constructor, and assignment operator are used.

ccOCVMaxParagraphSearchKeySets

lineKeySets

```
const ccOCVMaxLineSearchKeySets &lineKeySets(c_Int32
  index) const;

void lineKeySets(c_Int32 index, const
  ccOCVMaxLineSearchKeySets &keys);
```

- ```
  const ccOCVMaxLineSearchKeySets &lineKeySets(c_Int32
    index) const;
  ```

  Gets a single line-keysets object.

  Parameters

  index The line-keyset object index.

  Throws

  ccOCVMaxDefs::BadParams
  index is out of range.

- ```
  void lineKeySets(c_Int32 index, const
    ccOCVMaxLineSearchKeySets &keys);
  ```

  Sets a single line-keysets object.

  Parameters

  index The line-keyset object index.

  Throws

  ccOCVMaxDefs::BadParams
  index is out of range.

positionKeySets

```
const ccOCVMaxKeySet &positionKeySets(c_Int32 lineIndex,
  c_Int32 positionIndex) const;

void positionKeySets(c_Int32 lineIndex, c_Int32
  positionIndex, const ccOCVMaxKeySet &keys);
```

- ```
  const ccOCVMaxKeySet &positionKeySets(c_Int32 lineIndex,
    c_Int32 positionIndex) const;
  ```

  Gets a single positional keyset object. This function is a convenient way to "drill down" and get a single value deep in a hierarchy.

  Parameters

  lineIndex The index of the line.
  ```

positionIndex The index of the position.

Throws

ccOCVMaxDefs::BadParams
Any of the indices are out of range.

- `void positionKeySets(c_Int32 lineIndex, c_Int32 positionIndex, const ccOCVMaxKeySet &keys);`

Sets a single positional keyset object. This function is a convenient way to "drill down" and set a single value deep in a hierarchy.

Parameters

lineIndex The index of the line.

positionIndex The index of the position.

Notes

The setter requires that the all indices are valid. This function cannot be used to add new elements to a SearchKeySets object. For example, you cannot set position 6 on a line with only 3 position elements.

Throws

ccOCVMaxDefs::BadParams
Any of the indices are out of range.

appendLineKeySets

```
void appendLineKeySets(const ccOCVMaxLineSearchKeySets &lineKeySets);
```

Adds a single line-keyset object to the end of the paragraph.

numLineKeySets

```
c_Int32 numLineKeySets() const;
```

Returns the number of line keySets stored.

Operators

operator==

```
bool operator==(const ccOCVMaxParagraphSearchKeySets& other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxParagraphSearchKeySets&
  other) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxParagraphTrainParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxParagraphTrainParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxParagraphTrainParams** class encapsulates all of the parameters for a particular paragraph within an arrangement that are required for character verification.

Constructors/Destructors

Notes

The default constructor, destructor, copy constructor, and assignment operator are used.

Operators

operator==

```
bool operator==(const ccOCVMaxParagraphTrainParams& other)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxParagraphTrainParams& other)
  const;
```

Returns true if this object is not equal to the specified object.

Parameters

other	The object with which to compare for inequality.

ccOCVMaxParagraphTuneParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxParagraphTuneParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The OCVMax paragraph tune parameters specify which aspects of a given paragraph within an arrangement should be modified by tuning.

Enumerations

TuneFlags `enum TuneFlags;`

Enumeration for the tune flags.

Value	Meaning
eTuneNone = 0x0000	Does not tune anything.
eTuneSpotSizeFactor = 0x0001	These flags each tune the corresponding value of the **ccOCVMaxParagraph** object.
eTuneTrackingX = 0x0002	
eTuneTrackingY = 0x0004	
eTuneExtraLeadingX = 0x0008	
eTuneExtraLeadingY = 0x0010	
eTuneExtraStrokeWidth = 0x0020	
eTuneSpotSpacingXScale = 0x0040	
eTuneSpotSpacingYScale = 0x0080	
eTunePolarity = 0x0100	

Value	Meaning
eTunePoseUniformScale = 0x0200	These flags each tune the corresponding value of the arrangement's **paragraphPose()** for the paragraph.
eTunePoseAspect = 0x0400	
eTunePoseRotation = 0x0800	
eTunePoseShear = 0x1000	
eTuneAll = (*eTuneSpotSizeFactor* | *eTuneTrackingX* | *eTuneTrackingY* | *eTuneExtraLeadingX* | *eTuneExtraLeadingY* | *eTuneExtraStrokeWidth* | *eTuneSpotSpacingXScale* | *eTuneSpotSpacingYScale* | *eTunePolarity* | *eTunePoseUniformScale* | *eTunePoseAspect* | *eTunePoseRotation* | *eTunePoseShear*)	All values are tuned.
eTuneDefault = (*eTuneAll* & ~(*eTuneTrackingY* | *eTuneExtraLeadingX* | *eTunePoseAspect* | *eTunePoseShear* | *eTuneSpotSpacingXScale* | *eTuneSpotSpacingYScale* | *eTuneExtraStrokeWidth*))	The default values corresponding to the specified flags are tuned.

Constructors/Destructors

ccOCVMaxParagraphTuneParams

```
ccOCVMaxParagraphTuneParams();
```

Constructs a default tune parameters object.

Notes

The default copy constructor, assign operator, and destructor are used.

Public Member Functions

flags

```
c_UInt32 flags() const;

void flags(c_UInt32 f);
```

- `c_UInt32 flags() const;`

  Gets a bitfield of TuneFlags specifying what to tune.

- `void flags(c_UInt32 f);`

  Sets a bitfield of TuneFlags specifying what to tune.

  The default value is *eTuneDefault*.

  Parameters

f	The bitfield value of TuneFlags.

ccOCVMaxParagraphTuneParams

ccOCVMaxPositionResult

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxPositionResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The OCVMax Position Result class contains all the result information for an instance of a single character position within a particular line within a particular paragraph within a particular arrangement. This information includes the status of the search at that position (verified, confused, or failed), the key of the detected character (if any), the pose of the character in various coordinate systems, the score of the character, the keys and scores for any characters that may be confused with the found character, and the confidence score, which is the score of the found character minus the score of the highest-scoring confusion character (but not less than zero).

Constructors/Destructors

ccOCVMaxPositionResult

```
ccOCVMaxPositionResult();
```

Constuct a result object with a failed status.

Notes

The default destructor, copy constructor, and assignment operators are used.

Public Member Functions

key

```
c_Int32 key() const;
```

Returns the key of the character found at this position.

Notes

Returns -1 if status is *eFailed* or if the verification type for the search was *eIgnore*.

ccOCVMaxPositionResult

status

```
ccOCVMaxDefs::PositionStatus status() const;
```

Returns the verification status: verified, confused, or failed.

VerificationType *eNormal*:

Standard verification. Returns failed if no element of the key set had a score above the accept threshold. Returns verified if an element of the key set had a score above the accept threshold, and its confidence score (the difference between its score and that of its highest-scoring confusable key) is above the confidence threshold. Otherwise returns confused.

VerificationType *eAlwaysVerified*:

Standard verification. Returns verified regardless of the scores. Scores are computed.

VerificationType *eIgnore*:

No verification. Returns verified regardless of the scores. Scores are not computed, they are set to 1.0.

foundPose

```
bool foundPose() const;
```

Returns whether a pose was determined for this result.

clientPose

```
const cc2Xform& clientPose() const;
```

Returns the pose of the found character in client coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundPose()** operation returned false.

foundParagraphPose

```
bool foundParagraphPose() const;
```

Returns whether a paragraph pose was determined for this result.

paragraphPose

```
const cc2Xform& paragraphPose() const;
```

Returns the pose of the found character in paragraph coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundPose()** operation returned false.

foundArrangementPose

```
bool foundArrangementPose() const;
```

Returns whether an arrangement pose was determined for this result.

arrangementPose

```
const cc2Xform& arrangementPose() const;
```

Returns the pose of the found character in arrangement coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundPose()** operation returned false.

score

```
double score() const;
```

Returns the verification score of the found character. If the verification type is *eIgnore*, this will be 1.0.

matchScore

```
double matchScore() const;
```

Returns the match component of the verification score.

zoneScore

```
double zoneScore() const;
```

Returns the zone component of the verification score.

fitError

```
double fitError() const;
```

Returns the *fitError* of the found character.

fitError ranges from 0.0 (best) to infinity (worst).

Notes

Value will be -1.0 if not available.

Although fitError has no theoretical upper limit, it is typically less than 10 in practical use.

isEmpty

```
bool isEmpty() const;
```

Returns whether the position is empty; that is, failed and with no match or zone violation information.

keyZoneViolations

```
c_UInt32 keyZoneViolations() const;
```

Returns which zones of the key search parameters were violated and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

fullZoneViolations

```
c_UInt32 fullZoneViolations() const;
```

Returns which zones of the full (that is, image or start pose) search params were violated and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

confusionKeys

```
const cmStd vector<c_Int32>& confusionKeys() const;
```

Returns the keys of the characters causing confusion, if the status is *eConfused*. Otherwise the vector is empty. The vector contains the confusing characters in order from most confusing (highest-scoring) to least confusing (lowest scoring).

confusionMatchScores

```
const cmStd vector<double>& confusionMatchScores() const;
```

Returns the confusion match scores corresponding to the *confusionKeys*, if the status is *eConfused*. Otherwise the vector is empty.

confidenceScore

```
double confidenceScore() const;
```

Returns the confidence score, which is the score minus the score of the best *confusionMatchScore*, but not less than zero. If the verification type is *eIgnore*, this will be 1.0.

hasResultRegion

```
bool hasResultRegion() const;
```

Gets whether the result contains a result region. See **resultRegion()** below.

resultRegion

```
ccAffineRectangle resultRegion() const;
```

Return the affine rectangle that describes the boundary of the matched character in the runtime image, in client coordinates.

Throws

ccOCVMaxDefs::NotFound
The **hasResultRegion()** operation returned false.

Operators

operator==

```
bool operator==(const ccOCVMaxPositionResult& other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxPositionResult& other) const;
```

Return true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxPositionResult

ccOCVMaxPositionResultStats

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxPositionResultStats;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxPositionResultStats** class computes cumulative statistics over a set of position results. These statistics can be helpful for determining optimal score accept thresholds for later runs or for process control.

Constructors/Destructors

ccOCVMaxPositionResultStats

```
ccOCVMaxPositionResultStats();
```

Constructs a default position result statistics object.

Notes

The default copy constructor, assign operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Resets the statistics.

add

```
void add(const ccOCVMaxPositionResult& result);
```

Adds the position result to the set of statistics.

Parameters

result	The statistical result.

numPositionResults

```
c_Int32 numPositionResults() const;
```

Gets the number of position results that have been added.

numVerified `c_Int32 numVerified() const;`

Gets the number of position results whose status was *eVerified*.

numConfused `c_Int32 numConfused() const;`

Gets the number of position results whose status was *eConfused*.

numFailed `c_Int32 numFailed() const;`

Gets the number of position results whose status was *eFailed*.

numNonFailed `c_Int32 numNonFailed() const;`

Gets the number of position results whose status was not *eFailed*; that is, the number of position results that were either *eVerified* or *eConfused*.

hasScores `bool hasScores() const;`

Gets whether any score statistics are available.

minNonFailedScore

`double minNonFailedScore() const;`

Gets the minimum score of all the position results whose status was not *eFailed*. Typically, one should expect that setting a score accept threshold somewhat lower than this value would allow finding all such results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasScores()** operation returned false.

maxNonFailedScore

`double maxNonFailedScore() const;`

Gets the maximum score of all the position results whose status was not *eFailed*.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasScores()** operation returned false.

meanNonFailedScore

`double meanNonFailedScore() const;`

Gets the mean score of all the position results whose status was not *eFailed*.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasScores()** operation returned false.

ccOCVMaxPositionResultStats

ccOCVMaxProgress

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxProgress;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxProgress** class provides information about how much progress has been made by OCVMax towards completion of a potentially time-consuming task such as training or tuning a tool.

Enumerations

Type

```
enum Type;
```

The type of operation whose progress is being reported.

Value	Meaning
eTrain	Progress is for **ccOCVMaxTool::train()**
eTune	Progress is for **ccOCVMaxTool::tune()**

Constructors/Destructors

ccOCVMaxProgress

```
ccOCVMaxProgress(): progress_(0.), userParams_(NULL) {}
```

Constructs a progress object.

Notes

The default copy constructor, assign operator, and destructor are used.

Public Member Functions

type

```
Type type() const;
void type(Type t);
```

- `Type type() const;`

  Gets the type of function whose progress is measured.

- `void type(Type t);`

  Sets the type of function whose progress is measured.

  Parameters

  t — The function type. Must be one of:
  ccOCVMaxProgress::eTrain
  ccOCVMaxProgress::eTune

progress

```
double progress() const;
void progress(double p);
```

- `double progress() const;`

  Gets progress, in the range [0, 1].

- `void progress(double p);`

  Sets progress, in the range [0, 1].

  The task has just started when progress is 0, and the task is complete when the progress is 1.

  Parameters

  p — The progress value.

  Throws

  ccOCVMaxDefs::BadParams
  Progress value is less than 0 or greater than 1.

  The default progress value is 0.

userParams

```
const void* userParams() const;
void* userParams();
void userParams(void* u);
```

- `const void* userParams() const;`

  Gets the user parameters.

- `void* userParams();`

  Sets the user parameters.

- `void userParams(void* u);`

  Sets the user parameters.

  Typically, the user parameters are the parameters passed as the *userParams* to the tool's **train()** or **tune()** function. As an example, this might be a GUI object that implements a progress bar.

  Parameters

u	The user parameter.

  Notes

  The pointer is not owned by this object.

  userParams are not serialized.

  The default user parameters value is NULL.

ccOCVMaxProgress

ccOCVMaxProgressCallback

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxProgressCallback;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **ccOCVMaxProgressCallback** class allows the user to implement custom behavior based on how much progress has been made by OCVMax towards completion of a potentially time-consuming task such as training or tuning a tool. One example of callback function might be to provide a GUI progress bar whose state indicates the amount of progress made.

Constructors/Destructors

ccOCVMaxProgressCallback

```
ccOCVMaxProgressCallback(Function f = NULL, void*
  userParams = NULL): callback_(f), userParams_(userParams)
  {}

ccOCVMaxProgressCallback(const ccOCVMaxProgressCallback&
  rhs): callback_(NULL), userParams_(NULL) {}
```

- ```
  ccOCVMaxProgressCallback(Function f = NULL, void*
    userParams = NULL): callback_(f), userParams_(userParams)
    {}
  ```

  Constructs a progress callback object that will execute the specified function when OCVMax reports a progress update. A shallow copy of *userParams* will be contained in the **ccOCVMaxProgress** object passed to the callback function.

  Parameters

f	The specified function.
userParams	The user parameters.
  ```

- `ccOCVMaxProgressCallback(const ccOCVMaxProgressCallback& rhs): callback_(NULL), userParams_(NULL) {}`

  Copy constructor; copies actually nothing, since there would usually be lifetime issues.

  Parameters

  rhs The source of the copy.

  Notes

  The default destructor is used.

Operators

operator=

```
ccOCVMaxProgressCallback& operator=(const
 ccOCVMaxProgressCallback& rhs);
```

Assignment operator; assigns actually nothing, since there would usually be lifetime issues.

Parameters

rhs The assignment source.

Public Member Functions

callback

```
void callback(Function f);

Function callback() const;
```

- `void callback(Function f);`

  Sets the function to be used.

  Parameters

  f The function to be used.

- `Function callback() const;`

  Gets the function to be used.

  The default function value is NULL.

userParams

```
const void* userParams() const;

void* userParams();

void userParams(void* u);
```

- `const void* userParams() const;`

  Gets the user parameters, which will be passed to the callback function whenever it is invoked.

- `void* userParams();`

  Sets the user parameters, which will be passed to the callback function whenever it is invoked.

- `void userParams(void* u);`

  Sets the user parameters, which will be passed to the callback function whenever it is invoked.

  Parameters

u	The new user parameters.

  Notes

  The pointer is not owned by this object. The user is responsible for ensuring that the object pointed to continues to exist for the entire duration of the train/tune call.

  The default user parameters value is NULL.

assignFull

```
void assignFull(const ccOCVMaxProgressCallback& rhs)
 {
  if (this != &rhs)
    {
     callback_ = rhs.callback_;
     userParams_ = rhs.userParams_;
    }
 }
```

Full assignment, typically for short-lived copies.

Typedefs

```
typedef c_Int32 (*Function)(ccOCVMaxProgress& progress);
```

A function pointer type for user-specified callback functions.

ccOCVMaxProgressCallback

ccOCVMaxResult

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxResult** class contains all the result information for a single application of the OCVMax tool.

Constructors/Destructors

ccOCVMaxResult

```
ccOCVMaxResult();
```

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

foundPose

```
bool foundPose() const;
```

Returns whether a pose was determined for this result.

clientPose

```
cc2XformBasePtrh_const clientPose() const;
```

Returns the pose of the arrangement in client coordinates.

Throws

ccOCVMaxDefs::NotFound
The **foundPose()** operation returned false.

foundPoseLinear

```
bool foundPoseLinear() const;
```

Returns whether a linear pose was determined for this result.

clientPoseLinear

```
const cc2XformLinear &clientPoseLinear() const;
```

Returns the pose of the arrangement in client coordinates, but linearized over the area covered by the arrangement.

Throws

ccOCVMaxDefs::NotFound
The **foundPoseLinear()** operator returned false.

score

```
double score() const;
```

Returns the overall score of the arrangement (the average of the scores of all of the paragraphs that the tool attempted to verify).

numParagraphs

```
c_Int32 numParagraphs() const;
```

Returns the total number of paragraphs.

numParagraphsVerified

```
c_Int32 numParagraphsVerified() const;
```

Returns the number of paragraphs correctly verified.

verified

```
bool verified() const;
```

Returns true if the tool passed (all character positions have been verified), and false otherwise.

numPositionsEmpty

```
c_Int32 numPositionsEmpty() const;
```

Returns the number of character positions that are empty.

keyZoneViolations

```
c_UInt32 keyZoneViolations() const;
```

Returns which zones of the key search params were violated by at least one position and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

fullZoneViolations

```
c_UInt32 fullZoneViolations() const;
```

Returns which zones of the full (i.e. image or start pose) search params were violated by at least one position and thus decreased the zone score, as a bitwise-OR of values from **ccOCVMaxDefs::DOF**.

paragraphResults

```
const cmStd vector<ccOCVMaxParagraphResult>&
  paragraphResults() const;
```

Returns the paragraph results for all paragraphs on which verification has been attempted.

timeoutOccurred

```
bool timeoutOccurred() const;
```

Gets whether a timeout occurred during the run.

Notes

The other information in the result represents the best result obtained as of the time when the timeout occurred; it is possible that a better result might have been obtained if the run had been allowed to run to completion without a timeout.

Operators

operator==

```
bool operator==(const ccOCVMaxResult& other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxResult& other) const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for inequality.

ccOCVMaxResult

ccOCVMaxResultDOFStats

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxResultDOFStats;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxResultDOFStats** class computes cumulative statistics over a set of run results. These statistics can be helpful for determining optimal DOF ranges for later runs or for process control.

Constructors/Destructors

ccOCVMaxResultDOFStats

```
ccOCVMaxResultDOFStats();
```

Constructs a default statistics object.

Notes

The default copy constructor, assignment operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Resets the statistics.

add

```
void add(const ccOCVMaxResult& result);
```

Adds the result to the set of statistics to measure the observed range of DOFs.

Parameters

result	The result to be added.

numResults

```
c_Int32 numResults() const;
```

Gets the number of results that have been added to the statistics.

numFullPositionResults

```
c_Int32 numFullPositionResults() const;
```

Gets the cumulative number of position results contained within all the results that have been added.

Notes

The term "full" is meant to indicate that the statistics for the position results affects the full initial search parameters for a run, which might be specified as either image search parameters or as start pose search parameters.

numKeyPositionResultPairs

```
c_Int32 numKeyPositionResultPairs() const;
```

Gets the cumulative number of pairs of position results contained within all the results that have been added. A pair of position results consists of two adjacent position results, which are thus representative of DOFs for key search parameters.

hasKeyStats

```
bool hasKeyStats() const;
```

Gets whether any statistics relevant for key search parameters are available.

keyUniformScaleRange

```
ccRange keyUniformScaleRange() const;
```

Gets the observed range of the specific DOF obtained by comparing the DOFs of adjacent position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the key search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyStats()** operation returned false.

keyXScaleRange

```
ccRange keyXScaleRange() const;
```

Gets the observed range of the specific DOF obtained by comparing the DOFs of adjacent position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the key search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyStats()** operation returned false.

keyYScaleRange

```
ccRange keyYScaleRange() const;
```

Gets the observed range of the specific DOF obtained by comparing the DOFs of adjacent position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the key search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyStats()** operation returned false.

keyAspectRange

```
ccRange keyAspectRange() const;
```

Gets the observed range of the specific DOF obtained by comparing the DOFs of adjacent position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the key search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyStats()** operation returned false.

keyRotationRange

```
ccAngleRange keyRotationRange() const;
```

Gets the observed range of the specific DOF obtained by comparing the DOFs of adjacent position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the key search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyStats()** operation returned false.

keyShearRange

```
ccAngleRange keyShearRange() const;
```

Gets the observed range of the specific DOF obtained by comparing the DOFs of adjacent position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the key search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyStats()** operation returned false.

hasKeyXyStats

```
bool hasKeyXyStats() const;
```

Gets whether pairwise xy (translation) statistics are available.

keyMaxXyDiff

```
cc2Vect keyMaxXyDiff() const;
```

Gets the maximum xy (translation) difference between a position result and its expected position, where the expected position is based on an adjacent position result and the advance. This value can be useful for setting the **xyUncertainty()** of key search run parameters.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasKeyXyStats()** operation returned false.

hasFullStats

```
bool hasFullStats() const;
```

Gets whether full (image or start pose) statistics are available.

fullUniformScaleRange

```
ccRange fullUniformScaleRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullXScaleRange

```
ccRange fullXScaleRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullYScaleRange

```
ccRange fullYScaleRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullXyScaleRatioRange

```
ccRange fullXyScaleRatioRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullAspectRange

```
ccRange fullAspectRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullRotationRange

```
ccAngleRange fullRotationRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullShearRange

```
ccAngleRange fullShearRange() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

fullXyRect

```
ccRect fullXyRect() const;
```

Gets the observed range of the specific DOF obtained over all position results. Typically, one should expect that setting a DOF range slightly larger than the observed range in the full (that is, image or start pose) search run parameters would allow finding all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasFullStats()** operation returned false.

ccOCVMaxResultStats

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxResultStats;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The OCVMax result statistics object contains statistics computed over a set of results. The statistics include scores, DOF ranges, and status values.

Constructors/Destructors

ccOCVMaxResultStats

```
ccOCVMaxResultStats();
```

Constructs a default result statistics object.

Notes

The default copy constructor, assign operator, and destructor are used.

Public Member Functions

reset

```
void reset();
```

Resets the statistics.

add

```
void add(const ccOCVMaxResult& result);
```

Adds the result to the set of statistics to measure the observed range of DOFs.

Parameters

result	The result to be added.

numResults

```
c_Int32 numResults() const;
```

Gets the number of results that were added.

ccOCVMaxResultStats

numVerified `c_Int32 numVerified() const;`

Gets the number of results that were verified.

numFailed `c_Int32 numFailed() const;`

Gets the number of results that were not verified.

numTimedOut `c_Int32 numTimedOut() const;`

Gets the number of results that timed out.

hasScores `bool hasScores() const;`

Gets whether any score statistics are available.

minScore `double minScore() const;`

Gets the minimum score over all the results.

Throws
ccOCVMaxDefs::StatisticNotComputed
The **hasScores()** operation returned false.

maxScore `double maxScore() const;`

Gets the maximum score over all the results.

Throws
ccOCVMaxDefs::StatisticNotComputed
The **hasScores()** operation returned false.

meanScore `double meanScore() const;`

Gets the mean score over all the results.

Throws
ccOCVMaxDefs::StatisticNotComputed
The **hasScores()** operation returned false.

dofStats `const ccOCVMaxResultDOFStats& dofStats() const;`

Gets the DOF statistics accumulated over all the results.

combinedPositionStats

```
const ccOCVMaxPositionResultStats& combinedPositionStats()
  const;
```

Gets the position result statistics accumulated over all the positions over all the results.

hasPositionStats

```
bool hasPositionStats() const;
```

Gets whether position result statistics are available.

numParagraphs

```
c_Int32 numParagraphs() const;
```

Gets the total number of paragraphs in all the results.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasPositionStats()** operation returned false.

numLines

```
c_Int32 numLines(c_Int32 paragraphIndex) const;
```

Gets the total number of lines in all the results.

Parameters

paragraphIndex The paragraph index.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasPositionStats()** operation returned false.

numPositions

```
c_Int32 numPositions(c_Int32 paragraphIndex, c_Int32
lineIndex) const;
```

Gets the total number of positions in all the results.

Parameters

paragraphIndex The paragraph index.

lineIndex The line index.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasPositionStats()** operation returned false.

positionStats

```
const ccOCVMaxPositionResultStats& positionStats(c_Int32
paragraphIndex, c_Int32 lineIndex, c_Int32 positionIndex)
const;
```

Gets the statistics computed over only those position results at the specified paragraph index, line index, and position index.

Parameters

paragraphIndex The paragraph index.

lineIndex The line index.

positionIndex The position index.

Throws

ccOCVMaxDefs::StatisticNotComputed
The **hasPositionStats()** operation returned false.

dump

```
void dump(ccCvlOStream& out, c_Int32 indent = 0) const;
```

Outputs a text representation of all the statistics to the stream.

Parameters

out The stream to which the descriptive text is output.

indent The indent.

Throws

ccOCVMaxDefs::BadParams
indent is less than 0.

ccOCVMaxRunParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxRunParams** class encapsulates all of the OCVMax tool run-time parameters required for running the tool.

Constructors/Destructors

ccOCVMaxRunParams

```
ccOCVMaxRunParams (
  const cmStd vector<ccOCVMaxParagraphRunParams>&
                     paragraphParams = cmStd
                     vector<ccOCVMaxParagraphRunParams>(),
  const ccOCVMaxSearchRunParams &imageSearchRunParams =
                                  ccOCVMaxSearchRunParams(),
  const ccOCVMaxSearchRunParams &keySearchRunParams =
                                  ccOCVMaxSearchRunParams(),
  const ccOCVMaxSearchRunParams &startPoseSearchRunParams =
                                ccOCVMaxSearchRunParams());

ccOCVMaxRunParams
  (const ccOCVMaxArrangement &arr,
  const ccOCVMaxSearchRunParams &imageSearchRunParams =
                                  ccOCVMaxSearchRunParams(),
  const ccOCVMaxSearchRunParams &keySearchRunParams =
                                  ccOCVMaxSearchRunParams(),
  const ccOCVMaxSearchRunParams &startPoseSearchRunParams =
                                ccOCVMaxSearchRunParams());
```

-
```
ccOCVMaxRunParams (
  const cmStd vector<ccOCVMaxParagraphRunParams>&
                     paragraphParams = cmStd
                     vector<ccOCVMaxParagraphRunParams>(),
  const ccOCVMaxSearchRunParams &imageSearchRunParams =
                                  ccOCVMaxSearchRunParams(),
```

```
  const ccOCVMaxSearchRunParams &keySearchRunParams =
                                    ccOCVMaxSearchRunParams(),
  const ccOCVMaxSearchRunParams &startPoseSearchRunParams =
                                  ccOCVMaxSearchRunParams());
```

Constructs a runtime parameter object using the supplied parameters.

Parameters

paragraphParams
Paragraph parameters.

imageSearchRunParams
Image search run-time parameters.

keySearchRunParams
Key search run-time parameters.

startPoseSearchRunParams
Start pose search run-time parameters.

- ```
  ccOCVMaxRunParams(
    const ccOCVMaxArrangement &arr,
    const ccOCVMaxSearchRunParams &imageSearchRunParams =
                                    ccOCVMaxSearchRunParams(),
    const ccOCVMaxSearchRunParams &keySearchRunParams =
                                    ccOCVMaxSearchRunParams(),
    const ccOCVMaxSearchRunParams &startPoseSearchRunParams =
                                  ccOCVMaxSearchRunParams());
  ```

  Constructs with a default-constructed paragraph parameters for each paragraph in the given arrangement.

  Parameters

  arr The arrangement.

  imageSearchRunParams
  Image search run-time parameters.

  keySearchRunParams
  Key search run-time parameters.

  startPoseSearchRunParams
  Start pose search run-time parameters.

  Notes

  The default destructor, copy constructor, and assignment operator are used.
  ```

Public Member Functions

imageSearchRunParams

```
const ccOCVMaxSearchRunParams& imageSearchRunParams()
  const;

void imageSearchRunParams
  (const ccOCVMaxSearchRunParams &searchRunParams);
```

- ```
  const ccOCVMaxSearchRunParams& imageSearchRunParams()
    const;
  ```

  Gets the search run-time parameters to use for pattern alignment in the image.

- ```
  void imageSearchRunParams
    (const ccOCVMaxSearchRunParams &searchRunParams);
  ```

  Sets the search run-time parameters to use for pattern alignment in the image.

  Parameters

  searchRunParams Search run-time parameters.

keySearchRunParams

```
const ccOCVMaxSearchRunParams& keySearchRunParams() const;

void keySearchRunParams
  (const ccOCVMaxSearchRunParams &keySearchRunParams);
```

- ```
  const ccOCVMaxSearchRunParams& keySearchRunParams() const;
  ```

  Gets the search run-time parameters to use for alignment of each keyset relative to its neighbor.

  Throws

  ccOCVMaxDefs::BadParams

  The *keySearchRunParams* value has an angle DOF for which it is not true that low is less than or equal to 0 and high is greater than or equal to 0; or the *keySearchRunParams* value has a scale DOF for which it is not true that low is less than or equal to 1 and high is greater than or equal to 1. In other words, the "neutral" DOF value must be within the specified range.
  ```

- ```
  void keySearchRunParams
    (const ccOCVMaxSearchRunParams &keySearchRunParams);
  ```

  Sets the search run-time parameters to use for alignment of each keyset relative to its neighbor.

  Parameters

  keySearchRunParams Key search run-time parameters.

  Throws

  ccOCVMaxDefs::BadParams

  The *keySearchRunParams* value has an angle DOF for which it is not true that low is less than or equal to 0 and high is greater than or equal to 0; or the *keySearchRunParams* value has a scale DOF for which it is not true that low is less than or equal to 1 and high is greater than or equal to 1. In other words, the "neutral" DOF value must be within the specified range.

startPoseSearchRunParams

```
const ccOCVMaxSearchRunParams& startPoseSearchRunParams()
  const;

void startPoseSearchRunParams
  (const ccOCVMaxSearchRunParams
  &startPoseSearchRunParams);
```

- ```
  const ccOCVMaxSearchRunParams& startPoseSearchRunParams()
    const;
  ```

  Gets the search run-time parameters to use for pattern alignment in the image when a *startPose* is specified at run-time.

- ```
  void startPoseSearchRunParams
    (const ccOCVMaxSearchRunParams
    &startPoseSearchRunParams);
  ```

  Sets the search run-time parameters to use for pattern alignment in the image when a *startPose* is specified at run-time.

  Parameters

  startPoseSearchRunParams

  The start pose search run-time parameters.
  ```

paragraphParams

```
const cmStd vector<ccOCVMaxParagraphRunParams>&
  paragraphParams() const;

void paragraphParams(const cmStd
  vector<ccOCVMaxParagraphRunParams>& params);
```

- `const cmStd vector<ccOCVMaxParagraphRunParams>& paragraphParams() const;`

  Gets the run-time parameters for the paragraphs to be verified.

- `void paragraphParams(const cmStd vector<ccOCVMaxParagraphRunParams>& params);`

  Sets the run-time parameters for the paragraphs to be verified.

  Parameters

  params The run-time parameters.

earlyAcceptThreshold

```
double earlyAcceptThreshold() const;

void earlyAcceptThreshold(double eat);
```

- `double earlyAcceptThreshold() const;`

  Gets the threshold for early acceptance of a potential line match as the true instance of the line in the image.

- `void earlyAcceptThreshold(double eat);`

  Sets the threshold for early acceptance of a potential line match as the true instance of the line in the image.

  If the percentage of keys in the line that match (that is, are verified or confused) is above this threshold, no further searching for the line is performed. This speeds up the search process when the user knows that there will not be spurious characters in the run-time image (for example, when it is known that the only text in the image will be the string to be verified).

  The default value is 0.75.

Parameters

eat Early acceptance threshold.

Throws

ccOCVMaxDefs::BadParams
The setter is given a value not between 0 and 1.

earlyFailThreshold

```
double earlyFailThreshold() const;
void earlyFailThreshold(double eft);
```

- `double earlyFailThreshold() const;`

  Gets the threshold for early failure of a potential line match as the true instance of the line in the image.

- `void earlyFailThreshold(double eft);`

  Sets the threshold for early failure of a potential line match as the true instance of the line in the image.

  If the percentage of keys in the line that match (that is, are verified or confused) is below this threshold, the section of the image containing the potential line will not be considered any further. This speeds up the search process when the user knows that the image should be clean, with no missing or badly damaged characters allowed.

  The default value is 1.0 (that is, stop as soon as one character fails).

  Parameters

  eft Early failure threshold.

  Throws

  ccOCVMaxDefs::BadParams
  The setter is given a value not between 0 and 1.

computePoses

```
c_UInt32 computePoses() const;
void computePoses(c_UInt32 sel);
```

- `c_UInt32 computePoses() const;`

  Gets which result poses to compute, as a bitwise-OR of values from **ccOCVMaxDefs::PoseCompute**.

- `void computePoses(c_UInt32 sel);`

  Sets which result poses to compute, as a bitwise-OR of values from **ccOCVMaxDefs::PoseCompute**.

  Computing poses takes additional execution time and may not be needed in all applications; *ePoseComputeArrangementNonlinear* in particular can be expensive.

  Note that the poses of individual character positions are always computed. Poses that are not computed will cause the relevant **foundPose()** getter to return false and the relevant pose getters to throw **ccOCVMaxDefs::NotFound()**.

  The following table shows which bits must be set in order to make a given type of pose result valid.

  Bits:

  ePoseComputeLineLinear = L

  ePoseComputeParagraphLinear = P

  ePoseComputeArrangementLinear = A

  ePoseComputeArrangementNonlinear = AN

  Flags are set (X = valid).

	Bits			
Pose result type	L	P	A	AN
positionResult.clientPose				
positionResult.paragraphPose		X		
positionResult.arrangementPose			X	
lineResult.clientPose	X			
lineResult.paragraphPose	X	X		
lineResult.arrangementPose	X		X	
paragraphResult.clientPose		X		
paragraphResult.arrangementPose		X	X	

Pose result type	Bits L	P	A	AN
result.clientPoseLinear			X	
result.clientPose				X

Parameters

sel

The default value is 0 (only position result poses computed).

Throws

ccOCVMaxDefs::BadParams

sel is not a valid bitwise-OR of values from **ccOCVMaxDefs::PoseCompute**.

nonlinearXformType

```
ccOCVMaxDefs::NonlinearXformType nonlinearXformType()
  const;

void nonlinearXformType(ccOCVMaxDefs::NonlinearXformType
  type);
```

- `ccOCVMaxDefs::NonlinearXformType nonlinearXformType()
  const;`

  Gets the type of xform to use when computing the nonlinear arrangement pose.

- `void nonlinearXformType(ccOCVMaxDefs::NonlinearXformType
  type);`

  Sets the type of xform to use when computing the nonlinear arrangement pose.

  Parameters

  type Type of xform.

Operators

operator== `bool operator==(const ccOCVMaxRunParams& other) const;`

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!= `bool operator!=(const ccOCVMaxRunParams& other) const;`

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxRunParams

ccOCVMaxSearchRunParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxSearchRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxSearchRunParams** class encapsulates all the parameters used to run the arrangement and/or character searches during the course of processing. This object is conceptually similar to a **ccPMAlignRunParams** object in how it is used to control the search space.

Constructors/Destructors

ccOCVMaxSearchRunParams

```
ccOCVMaxSearchRunParams();
```

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

acceptThreshold

```
double acceptThreshold() const;

void acceptThreshold(double a);
```

- `double acceptThreshold() const;`

  Gets the threshold used to locate character shapes in the run-time image.

- `void acceptThreshold(double a);`

  Sets the threshold used to locate character shapes in the run-time image.

  The default value is 0.5.

Parameters

a The threshold value.

Throws

ccOCVMaxDefs::BadParams
The setter is given a value not between 0 and 1.

contrastThreshold

```
double contrastThreshold() const;

void contrastThreshold(double c);
```

- `double contrastThreshold() const;`

  Gets the amount of contrast (in run-time image grey levels) that a result must have to be accepted.

- `void contrastThreshold(double c);`

  Sets the amount of contrast (in run-time image grey levels) that a result must have to be accepted.

  The default value is 10.

  Parameters

  c The contrast amount.

  Throws

  ccOCVMaxDefs::BadParams
  The setter is given a negative value.

xyUncertainty

```
cc2Vect xyUncertainty() const;

void xyUncertainty(const cc2Vect& xyUnc);
```

- `cc2Vect xyUncertainty() const;`

  Gets the uncertainty of the starting pose in the x and y dimensions, in client coordinates.

- `void xyUncertainty(const cc2Vect& xyUnc);`

  Sets the uncertainty of the starting pose in the x and y dimensions, in client coordinates.

  This value has no effect when used in the *imageSearchRunParams* parameter of **ccOCVMaxRunParams**. It only has an effect on the keySearchRunParams parameter and the *startPoseSearchRunParams* parameter.

  The default value is 5,5.

  Parameters

  xyUnc The uncertainty of the starting pose in the x and y dimensions.

  Throws

  ccOCVMaxDefs::BadParams
  Either element of the *cc2Vect* is negative.

zoneEnable

```
c_UInt32 zoneEnable() const;
void zoneEnable(c_UInt32 enable);
```

- `c_UInt32 zoneEnable() const;`

  Gets the zone enable for each degree of freedom.

- `void zoneEnable(c_UInt32 enable);`

  Sets the zone enable for each degree of freedom.

  Non-zero bits in the value indicates degrees of freedom to be searched. If *zoneEnable* is set for a given DOF, the search will be performed between that DOF's *zoneLow* and *zoneHigh* values; otherwise, it will use the single nominal value.

  Parameters

  enable The zone enable parameter.

  Requires that the value be 0 or a bitwise OR of values from the **ccOCVMaxDefs::DOF** enumeration.

  The default value is *ccOCVMaxDefs::eAngle* bitwise OR *ccOCVMaxDefs::eUniformScale*

nominal

```
double nominal(ccOCVMaxDefs::DOF dof) const;
void nominal(ccOCVMaxDefs::DOF dof, double val);
```

- `double nominal(ccOCVMaxDefs::DOF dof) const;`

  Gets the nominal value for the given DOF.

  Parameters

  dof The DOF.

- `void nominal(ccOCVMaxDefs::DOF dof, double val);`

  Sets the nominal value for the given DOF.

  The nominal value is only used if the given DOF is not enabled by **zoneEnable()**.

  Parameters

  dof The DOF.

  val The nominal value.

  Notes

  Angle and shear are specified in degrees.

  If this object is used in the *keySearchRunParams* parameter of **ccOCVMaxRunParams**, then the **ccOCVMaxRunParams** object will throw *BadParams* if for a disabled angle or shear DOF, nominal is not equal to 0; or if for a disabled scale DOF, nominal is not equal to 1. In other words, the "identity" DOF value must be the nominal value. default Angle: 0.0, Scales: 1.0, Shear: 0.0

  Throws

  ccOCVMaxDefs::BadParams
  The setter is given a zero or negative scale value.

zoneLow

```
double zoneLow(ccOCVMaxDefs::DOF dof) const;
```

Gets the low zone parameter for the given DOF.

Parameters

dof The DOF.

zoneHigh

```
double zoneHigh(ccOCVMaxDefs::DOF dof) const;
```

Gets the high zone parameter for the given DOF.

Parameters

dof The DOF.

zone

```
void zone(ccOCVMaxDefs::DOF dof, double low, double high);
```

Sets the low and high zone parameters for the given DOF.

Parameters

dof The DOF.

low The low zone parameter.

high The high zone parameter.

Notes

Angle and shear are specified in degrees, and the search is performed from the low value to the high value.

If this object is used in the *keySearchRunParams* parameter of **ccOCVMaxRunParams**, then the **ccOCVMaxRunParams** object will throw *BadParams* if for an enabled angle or shear DOF, it is not true that low is less than or equal to 0 and high is greater than or equal to 0, or if for an enabled scale DOF, it is not true that low is less than or equal to 1 and high is greater than or equal to 1. In other words, the "identity" DOF value must be within the specified range.

The default values: Angle: -5 to 5. Scales: 0.95 to 1.05. Shear: 0 to 0.

Throws

ccOCVMaxDefs::BadParams
The setter is given a zero or negative scale value, or *zoneLow* is greater than *zoneHigh* for a scale DOF.

---

zoneOverlap

```
double zoneOverlap(ccOCVMaxDefs::DOF dof) const;

void zoneOverlap(ccOCVMaxDefs::DOF dof, double
overlapThresh);
```

- `double zoneOverlap(ccOCVMaxDefs::DOF dof) const;`

Gets the required zone overlap to consider two instances to be the same result.

Parameters

dof The DOF.

- `void zoneOverlap(ccOCVMaxDefs::DOF dof, double overlapThresh);`

  Sets the required zone overlap to consider two instances to be the same result.

  Two results are considered overlapping if the overlap amount is greater than overlapThresh.

  Parameters

  dof The DOF.

  overlapThresh The required zone overlap.

  Notes

  Angle and shear are specified in degrees.

  The default values: Angle: 30. Scales: 1.4. Shear: 30.

  Throws

  ccOCVMaxDefs::BadParams
  The setter is given a scale value less than 1 or an angle value less than 0.

xyOverlap

```
double xyOverlap() const;

void xyOverlap(double overlapThresh);
```

- `double xyOverlap() const;`

  Gets the required xy overlap to consider two instances to be the same result.

- `void xyOverlap(double overlapThresh);`

  Sets the required xy overlap to consider two instances to be the same result.

  Two results are considered overlapping if the overlap amount is greater than *overlapThresh*.

  The default value is 0.8.

  Parameters

  overlapThresh The overlap threshold.

  Throws

  ccOCVMaxDefs::BadParams
  The setter is given a negative value or a value greater than 1.

Operators

operator==

```
bool operator==(const ccOCVMaxSearchRunParams& other)
  const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVMaxSearchRunParams& other)
  const;
```

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxSearchRunParams

ccOCVMaxTool

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxTool;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxTool** class contains all the methods necessary to train and run the OCVMax tool.

Constructors/Destructors

ccOCVMaxTool

```
ccOCVMaxTool();

ccOCVMaxTool(const ccOCVMaxTool &from);
```

- `ccOCVMaxTool();`

  Constructor.

- `ccOCVMaxTool(const ccOCVMaxTool &from);`

  Copy-constructor.

  Parameters

  from The source of the copy.

operator=

```
ccOCVMaxTool &operator=(const ccOCVMaxTool &from);
```

Assignment operator.

Parameters

from Source of the assignment.

The default destructor is used.

Public Member Functions

train

```
void train(
  const ccOCVMaxArrangement &arrangement,
  const cc2XformBase& clientFromImage,
  const ccOCVMaxTrainParams &trainParams,
  ccDiagObject* diagObject = 0,
  c_UInt32 diagFlags = 0);
```

Trains the tool using the supplied arrangement.

Parameters

arrangement The supplied arrangement.

clientFromImage The client space from image space transform.

trainParams Train parameters.

diagObject An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccOCVMaxDefs::BadParams
There are no paragraphs in the arrangement,
or any of the lines in the arrangement contain no characters besides wildcards and spaces.

ccOCVMaxDefs::BadKey
Any paragraph contains keys that are not present in that paragraph's font,
or any paragraph's *alphabetKeys* contain keys that are not present in that paragraph's font.

Notes

Part of the training process is converting fonts into "alphabets" for use by the lower level tools. Depending on the sizes of these alphabets (that is, the number of possible characters in the search process), the training time can be quite large.

untrain

```
void untrain();
```

Untrains the tool. Releases memory required for training.

Notes

No effect if **isTrained()** == false.

isTrained

```
bool isTrained() const;
```

Returns true if this tool is trained and false otherwise.

arrangement

```
const ccOCVMaxArrangement& arrangement() const;
```

Returns the trained arrangement to be verified by the tool.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

trainParams

```
const ccOCVMaxTrainParams& trainParams() const;
```

Returns the train parameters used to train this tool.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

trainClientFromImage

```
cc2XformBasePtrh_const trainClientFromImage() const;
```

Returns the train client from image xform used to train this tool.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

confusionMatrixKeys

```
void confusionMatrixKeys(
  c_Int32 paragraphIndex,
  cmStd vector<c_Int32>& searchKeys,
  cmStd vector<c_Int32>& confusionKeys) const;
```

Returns the set of keys for which confusion has been computed for the given paragraph.

Parameters

paragraphIndex The paragraph index.

searchKeys The search keys.

confusionKeys The confusion keys.

confusion

```
double confusion(
  c_Int32 paragraphIndex,
  c_Int32 searchKey,
  c_Int32 confusionKey) const;
```

Returns the computed confusion value between the two specified keys, in the range [0, 1]. Value will be negative if the tool did not compute the given confusion score.

Parameters

paragraphIndex The paragraph index.

searchKey The search keys.

confusionKey The confusion keys.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

cellRectKey

```
ccAffineRectangle cellRectKey(
  c_Int32 paragraphIndex,
  c_Int32 key,
  const cc2Xform& clientFromParagraph = cc2Xform()) const;
```

Returns the cell rect for the given key at the specified pose.

Parameters

paragraphIndex The paragraph index.

key The key.

clientFromParagraph
The client transformation of the paragraph.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

ccOCVMaxDefs::BadKey
The key was not trained.

tune

```
void tune(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxTuneParams& tuneParams,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  const cc2XformBasePtrh_const &startPose,
  ccOCVMaxTuneResult &result,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0);

void tune(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxTuneParams& tuneParams,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  ccOCVMaxTuneResult &result,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0);
```

-

```
void tune(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxTuneParams& tuneParams,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  const cc2XformBasePtrh_const &startPose,
  ccOCVMaxTuneResult &result,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0);
```

Tunes the tool using the supplied image, tune parameters, and run-time parameters, and places results in the supplied result object.

Parameters

image	The supplied image.
tuneParams	The supplied tune parameters.
runParams	The supplied run-time parameters.
keysets	The key sets.
startPose	The start pose.
result	The supplied result object in which results are placed.
**diagObject*	An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.
diagFlags	The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void tune(
    const ccPelBuffer_const<c_UInt8> &image,
    const ccOCVMaxTuneParams& tuneParams,
    const ccOCVMaxRunParams &runParams,
    const ccOCVMaxArrangementSearchKeySets &keysets,
    ccOCVMaxTuneResult &result,
    ccDiagObject *diagObject = 0,
    c_UInt32 diagFlags = 0);
  ```

  Tunes the tool using the supplied image, tune parameters, and runtime parameters, and places results in the supplied result object.

  Tuning modifies the tool's arrangement to more closely match the instance found in the image. The initial arrangement must be reasonably close match to allow tuning to improve the arrangement.

  Parameters

image	The supplied image.
tuneParams	The supplied tune parameters.
runParams	The supplied run-time parameters.
keysets	The key sets.
result	The supplied result object in which results are placed.
*diagObject	An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.
diagFlags	The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values:
  ```

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

The search is restricted to the specified search keysets. If the overload that specifies *startPose* is used, the search process begins with the assumption that the string will be found very close to the given pose. If this is indeed the case, the tool can tune much faster than when it needs to search the entire image first.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

ccOCVMaxDefs::BadImage
The image is unbound.

ccOCVMaxDefs::BadParams
The paragraph/line/position/keyset structure specified by the **ccOCVMaxArrangementSearchKeySets** object does not match the trained arrangement structure. For example, the keysets object must have the same number of **ccOCVMaxParagraphSearchKeysets** as the trained arrangement has paragraphs.

If the tool was trained with *characterRegistration* mode of *eCorrelation*, then the following throws may also occur:

Throws

ccOCVMaxDefs::DOFNotTrained

The run-time parameters enable a DOF that was not enabled during **train()**.

OR

Run-time parameters (together with the *startPose*, if any) specify a DOF nominal value other than the value specified during **train()**.

OR

The size of an enabled DOF range is larger than the size of the trained range for that DOF, in any of the relevant search parameters (image, start pose, or key).

OR

The relevant full (that is, image or start pose) search params (together with the start pose, if any) contains a DOF value outside the trained range.

ccOCVMaxDefs::StartPoseDOFNotTrained

The *startPose* contains a non-identity value for aspect and the training DOFs did not have a non-identity *xScale* or *yScale*.

If the tool was trained with *characterRegistration* mode of *eCorrelation*, then the following throws may also occur in a future release (but are not yet implemented):

Throws

ccOCVMaxDefs::DOFNotTrainedImageCoords

Run-time parameters (together with the *startPose*, if any) specify a DOF nominal value in image coords other than the value specified during **train()**.

OR

The relevant full (that is, image or start pose) search params (together with the start pose, if any) contains a DOF value in image coords outside the trained range.

ccOCVMaxDefs::StartPoseDOFNotTrainedImageCoords

The *startPose* in image coords contains a non-identity value for aspect and the training DOFs did not have a non-identity *xScale* or *yScale*.

Requires that *diagObject* point to a valid diagnostic object or be NULL.

tuneRunParams

```
void tuneRunParams(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxTuneParams& tuneParams,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  const cc2XformBasePtrh_const &startPose,
  ccOCVMaxTuneResult &result,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0);

void tuneRunParams(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxTuneParams& tuneParams,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  ccOCVMaxTuneResult &result,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0);
```

-
```
void tuneRunParams(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxTuneParams& tuneParams,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  const cc2XformBasePtrh_const &startPose,
  ccOCVMaxTuneResult &result,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0);
```

Tunes the run params using the supplied image, tune parameters, and run-time parameters, and places results in the supplied result object.

Parameters

image	The supplied image.
tuneParams	The supplied tune parameters.
runParams	The supplied run-time parameters.
keysets	The key sets.
startPose	The start pose.
result	The supplied result object in which results are placed.
**diagObject*	An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void tuneRunParams(
    const ccPelBuffer_const<c_UInt8> &image,
    const ccOCVMaxTuneParams& tuneParams,
    const ccOCVMaxRunParams &runParams,
    const ccOCVMaxArrangementSearchKeySets &keysets,
    ccOCVMaxTuneResult &result,
    ccDiagObject *diagObject = 0,
    c_UInt32 diagFlags = 0);
  ```

  Tunes the run params using the supplied image, tune parameters, and run-time parameters, and places results in the supplied result object.

  Runs parameter tuning is cumulative with previous calls to **tuneRunParams()**, so that previous images should also work with the new parameters; the state is stored in *result*. To make tuning non-cumulative, call **resetHistory()** on *result* before calling **tuneRunParams()**. The arrangement and initial params must be a reasonably close match to allow tuning to improve the run params.

  Parameters

  image The supplied image.

  tuneParams The supplied tune parameters.

  runParams The supplied run-time parameters.

  keysets The key sets.

  result The supplied result object in which results are placed.

  *diagObject An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.

  diagFlags The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values:
  ```

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

The search is restricted to the specified search keysets. If the overload that specifies *startPose* is used, the search process begins with the assumption that the string will be found very close to the given pose. If this is indeed the case, the tool can tune much faster than when it needs to search the entire image first.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

ccOCVMaxDefs::BadImage
The image is unbound.

ccOCVMaxDefs::BadParams
The paragraph/line/position/keyset structure specified by the **ccOCVMaxArrangementSearchKeySets** object does not match the trained arrangement structure. For example, the keysets object must have the same number of **ccOCVMaxParagraphSearchKeysets** as the trained arrangement has paragraphs.

OR

The **ccOCVMaxTuneParams** object has any "max" value less than its corresponding "min" value.

If the tool was trained with the *characterRegistration* mode of *eCorrelation*, then the following throws may also occur:

Throws

ccOCVMaxDefs::DOFNotTrained

The run-time parameters enable a DOF that was not enabled during **train()**.

OR

The run-time parameters (together with the *startPose*, if any) specify a DOF nominal value other than the value specified during **train()**.

OR

The size of an enabled DOF range is larger than the size of the trained range for that DOF, in any of the relevant search params (image, start pose, or key).

OR

The relevant full (that is, image or start pose) search params (together with the start pose, if any) contains a DOF value outside the trained range.

ccOCVMaxDefs::StartPoseDOFNotTrained

The *startPose* contains a non-identity value for aspect and the training DOFs did not have a non-identity *xScale* or *yScale*.

If the tool was trained with *characterRegistration* mode of *eCorrelation*, then the following throws may also occur in a future release (but are not yet implemented):

Throws

ccOCVMaxDefs::DOFNotTrainedImageCoords

Run-time parameters (together with the *startPose*, if any) specify a DOF nominal value in image coords other than the value specified during **train()**.

OR

The relevant full (that is, image or start pose) search params (together with the start pose, if any) contains a DOF value in image coords outside the trained range.

ccOCVMaxDefs::StartPoseDOFNotTrainedImageCoords

The *startPose* in image coords contains a non-identity value for aspect and the training DOFs did not have a non-identity *xScale* or *yScale*.

Requires that *diagObject* point to a valid diagnostic object or be NULL.

run

```
void run(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  const cc2XformBasePtrh_const &startPose,
  ccOCVMaxResult &results,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0) const;

void run(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  ccOCVMaxResult &results,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0) const;
```

-
```
void run(
  const ccPelBuffer_const<c_UInt8> &image,
  const ccOCVMaxRunParams &runParams,
  const ccOCVMaxArrangementSearchKeySets &keysets,
  const cc2XformBasePtrh_const &startPose,
  ccOCVMaxResult &results,
  ccDiagObject *diagObject = 0,
  c_UInt32 diagFlags = 0) const;
```

Runs the tool using the supplied image and run-time parameters, and places results in the supplied result object.

Parameters

image	The supplied image.
runParams	The supplied run-time parameters.
keysets	The key sets.
startPose	The start pose.
results	The supplied result object in which results are placed.
**diagObject*	An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.
diagFlags	The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults*

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void run(
    const ccPelBuffer_const<c_UInt8> &image,
    const ccOCVMaxRunParams &runParams,
    const ccOCVMaxArrangementSearchKeySets &keysets,
    ccOCVMaxResult &results,
    ccDiagObject *diagObject = 0,
    c_UInt32 diagFlags = 0) const;
  ```

  Runs the tool using the supplied image and run-time parameters, and places results in the supplied result object.

  The search is restricted to the specified search keysets. If the overload that specifies *startPose* is used, the search process begins with the assumption that the string will be found very close to the given pose. If this is indeed the case, the tool can run much faster than when it needs to search the entire image first.

  Parameters

image	The supplied image.
runParams	The supplied run-time parameters.
keysets	The key sets.
results	The supplied result object in which results are placed.
*diagObject	An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.
diagFlags	The optional diagnostic flags. diagFlags must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*
  ```

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

ccOCVMaxDefs::BadImage
The image is unbound.

ccOCVMaxDefs::BadParams
The paragraph/line/position/keyset structure specified by the **ccOCVMaxArrangementSearchKeySets** object does not match the trained arrangement structure. For example, the keysets object must have the same number of **ccOCVMaxParagraphSearchKeysets** as the trained arrangement has paragraphs.

ccOCVMaxDefs::NoAlignableChars
All of the positions in any line consist of only blanks, ignored key sets, and/or very poorly alignable characters (typically small punctuation such as periods).

If the tool was trained with the *characterRegistration* mode of *eCorrelation*, then the following throws may also occur:

Throws

ccOCVMaxDefs::DOFNotTrained
The run-time parameters enable a DOF that was not enabled during **train()**.

OR

The run-time parameters (together with the *startPose*, if any) specify a DOF nominal value other than the value specified during **train()**.

OR

The size of an enabled DOF range is larger than the size of the trained range for that DOF, in any of the relevant search params (image, start pose, or key).

OR

The relevant full (that is, image or start pose) search params (together with the start pose, if any) contains a DOF value outside the trained range.

ccOCVMaxDefs::StartPoseDOFNotTrained
The *startPose* contains a non-identity value for aspect and the training DOFs did not have a non-identity *xScale* or *yScale*.

If the tool was trained with characterRegistration mode of *eCorrelation*, then the following throws may also occur in a future release (but are not yet implemented):

Throws

ccOCVMaxDefs::DOFNotTrainedImageCoords
The run-time parameters (together with the *startPose*, if any) specify a DOF nominal value in image coords other than the value specified during **train()**.

OR

The relevant full (that is, image or start pose) search params (together with the start pose, if any) contains a DOF value in image coords outside the trained range.

ccOCVMaxDefs::StartPoseDOFNotTrainedImageCoords
The *startPose* in image coords contains a non-identity value for aspect and the training DOFs did not have a non-identity *xScale* or *yScale*.

Requires that *diagObject* point to a valid diagnostic object or be NULL.

draw

```
void draw(const ccOCVMaxResult &result, c_UInt32 drawFlags,
ccGraphicList &graphList) const;
```

Append graphics for the specified result to *graphList* according to the specified flags:

- *eDrawArrangement* : draw final arrangement pose.
- *eDrawParagraphs* : draw final paragraph poses.
- *eDrawPositions* : draw final character poses.
- *eDrawArrangementBox* : draw arrangement bounding box when drawing arrangement pose.
- *eDrawParagraphBoxes* : draw paragraph bounding boxes when drawing paragraph poses.
- *eDrawPositionBoxes* : draw character regions when drawing character poses.

Parameters

result The specified result.

drawFlags The specified flags.

graphList The graph list to which graphics is appended.

Notes

The following color scheme is used for poses and boxes:

Color	Status of Arrangement/Line/Position
Green	Passed.
Red	Failed.
Blue	Confused.

All graphics are drawn in client coordinates.

Throws

ccOCVMaxDefs::NotTrained
This tool is not currently trained.

Requires that the result specified was generated by calling **run()** from this tool in its current trained state.

operator==

```
bool operator==(const ccOCVMaxTool& other) const;
```

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other The object with which to compare for equality.

operator!= `bool operator!=(const ccOCVMaxTool& other) const;`

Returns true if this object is not equal to the specified object and false otherwise.

Parameters

other The object with which to compare for inequality.

ccOCVMaxTrainParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxTrainParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The OCVMax training parameters class encapsulates all of the parameters required to run the OCVMax Tool.

Constructors/Destructors

ccOCVMaxTrainParams

```
ccOCVMaxTrainParams();

ccOCVMaxTrainParams(const ccOCVMaxArrangement &arr);
```

- `ccOCVMaxTrainParams();`

  Notes

  The default destructor, copy constructor, and assignment operator are used.

- `ccOCVMaxTrainParams(const ccOCVMaxArrangement &arr);`

  Constructs with a default-constructed paragraph parameters for each paragraph in the given arrangement.

  Parameters

  arr The given arrangement.

Public Member Functions

paragraphParams

```
const cmStd vector<ccOCVMaxParagraphTrainParams>&
  paragraphParams() const;

void paragraphParams(
  const cmStd vector<ccOCVMaxParagraphTrainParams>&
  params);
```

- ```
  const cmStd vector<ccOCVMaxParagraphTrainParams>&
    paragraphParams() const;
  ```

  Gets the training parameters for the paragraphs to be verified.

  There must be one training parameter object for each paragraph in the arrangement, in order.

  Notes

  Training throws *BadParams* if the size of the vector is not the same as the number of paragraphs in the arrangement.

- ```
  void paragraphParams(
    const cmStd vector<ccOCVMaxParagraphTrainParams>&
    params);
  ```

  Sets the training parameters for the paragraphs to be verified.

  Parameters

  params The training parameters.

characterRegistration

```
ccOCVMaxDefs::CharacterRegistration
  characterRegistration() const;

void
characterRegistration(ccOCVMaxDefs::CharacterRegistration
  regType);
```

- ```
  ccOCVMaxDefs::CharacterRegistration
    characterRegistration() const;
  ```

  Gets the type of registration (fitting) done for each character found.
  ```

- ```
  void
  characterRegistration(ccOCVMaxDefs::CharacterRegistration
    regType);
  ```

  Sets the type of registration (fitting) done for each character found.

  - If *eStandard* is selected, the registration is done using PatMax technology with up to six degrees of freedom.
  - If *eCorrelation* is selected, the registration is done using the correlation technology with at most translation, rotation, and uniform scale. The maximum possible runtime ranges for rotation and uniform scale must be specified at training time.

  eStandard usually takes longer to run than *eCorrelation* in the cases where both modes apply (that is, at most rotation and uniform scale are used).

  The default value is *ccOCVMaxDefs::eStandard*

  Parameters

  regType The type of registration.

---

scoreMode

```
ccOCVMaxDefs::ScoreMode scoreMode() const;

void scoreMode(ccOCVMaxDefs::ScoreMode s);
```

- ```
  ccOCVMaxDefs::ScoreMode scoreMode() const;
  ```

  Gets the score mode used to compute the score for each character.

- ```
  void scoreMode(ccOCVMaxDefs::ScoreMode s);
  ```

  Sets the score mode used to compute the score for each character.

  - *ScoreUsingClutter* - reduces the character score if extra features (clutter) are present.
  - *ScoreUsingOptimizedClutter* - reduces the character score if extra features (clutter) are present, but optimizes score for relatively low-contrast clutter.
  - *ScoreWithoutClutter* - computes the score based only on the presence of expected features, ignoring any extra features (clutter).

  Parameters

  s The score mode.
  ```

Notes

ScoreUsingClutter is usually preferable to allow confusion to be computed more reliably, whereas *ScoreWithoutClutter* may be necessary in noisy images. Ignored if *characterRegistration* is *eCorrelation*.

The default value is *ScoreUsingOptimizedClutter*

zoneEnable

```
c_UInt32 zoneEnable() const;

void zoneEnable(c_UInt32 enable);
```

- `c_UInt32 zoneEnable() const;`

  Gets the zone enable for each degree of freedom.

- `void zoneEnable(c_UInt32 enable);`

  Sets the zone enable for each degree of freedom.

  Non-zero bits in the value indicates degrees of freedom that may be searched at runtime. If *zoneEnable* is set for a given DOF, the search will be performed between that DOF's *zoneLow* and *zoneHigh* values; otherwise, it will use the single nominal value.

  Parameters

  enable The zone enable.

  Requires that the value be 0 or a bitwise OR of values from the **ccOCVMaxDefs::DOF** enumeration.

  Notes

  This value is used only in *eCorrelation* registration mode. Other modes do not need to specify DOF ranges during training.

  The default value is
  ccOCVMaxDefs::eAngle bitwise OR *ccOCVMaxDefs::eUniformScale*

nominal

```
double nominal(ccOCVMaxDefs::DOF dof) const;

void nominal(ccOCVMaxDefs::DOF dof, double val);
```

- `double nominal(ccOCVMaxDefs::DOF dof) const;`

  Gets the nominal value for the given DOF.

  Parameters

  dof The given DOF.

- `void nominal(ccOCVMaxDefs::DOF dof, double val);`

  Sets the nominal value for the given DOF.

  The nominal value is only used if the given DOF is not enabled by **zoneEnable()**.

  Parameters

  dof The given DOF.

  val The nominal value.

  Notes

  Angle and shear are specified in degrees.
  This value is used only in the *eCorrelation* registration mode. Other modes do not need to specify DOF ranges during training.

  The default values: Angle: 0.0, Scales: 1.0, Shear: 0.0

  Throws

  ccOCVMaxDefs::BadParams
  The setter is given a zero or negative scale value.

zoneLow `double zoneLow(ccOCVMaxDefs::DOF dof) const;`

Gets the low zone parameters for the given DOF.

Parameters

dof The given DOF.

zoneHigh `double zoneHigh(ccOCVMaxDefs::DOF dof) const;`

Gets the high zone parameters for the given DOF.

Parameters

dof The given DOF.

zone `void zone(ccOCVMaxDefs::DOF dof, double low, double high);`

Sets the low and high zone parameters for the given DOF.

Parameters

dof The given DOF.

low The low zone parameter.

high The high zone parameter.

Notes

Angle and shear are specified in degrees, and the search is performed from the low value to the high value.

If this object is used in the *keySearchRunParams* parameter of **ccOCVMaxRunParams**, then the **ccOCVMaxRunParams** object will throw *BadParams* if for an enabled angle or shear DOF, it is not true that low is less than or equal to 0 and high is greater than or equal to 0; or if for an enabled scale DOF, it is not true that low is less than or equal to 1 and high is greater than or equal to 1. In other words, the "neutral" DOF value must be within the specified range.

This value is used only in *eCorrelation* registration mode. Other modes do not need to specify DOF ranges during training.

The default values: Angle: -10 to 10. Scales: 0.9 to 1.1. Shear: 0 to 0.

Throws

ccOCVMaxDefs::BadParams
The setter is given a zero or negative scale value, or *zoneLow* is greater than *zoneHigh* for a scale DOF.

---

callback

```
const ccOCVMaxProgressCallback& callback() const;
void callback(const ccOCVMaxProgressCallback& c);
```

- `const ccOCVMaxProgressCallback& callback() const;`

Gets a callback function for training.

- `void callback(const ccOCVMaxProgressCallback& c);`

Sets a callback function for training.

The callback function will be called at various points during the tool's **train()** function to indicate the amount of progress made to allow implementing functionality such as a progress bar.

The callback will initially be called with a progress value of 0.0, and near the end of training it will be called with a value of 1.0; it may be called with several increasing values in between.

Parameters

c The callback function.

Notes

The callback object is not serialized.

computeConfusionMatrix

```
bool computeConfusionMatrix() const;
void computeConfusionMatrix(bool c);
```

- `bool computeConfusionMatrix() const;`

  Gets whether to compute the confusion matrix during training.

- `void computeConfusionMatrix(bool c);`

  Sets whether to compute the confusion matrix during training.

  If the confusion matrix is not computed, then characters will never be considered to be confused. This parameter should be used only in unusual circumstances.

  Parameters

c	Boolean parameter: if true, the confusion matrix is calculated during training.

Operators

operator== `bool operator==(const ccOCVMaxTrainParams& other) const;`

Returns true if this object is equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for equality.

operator!= `bool operator!=(const ccOCVMaxTrainParams& other) const;`

Return true if this object is not equal to the specified object and false otherwise.

Parameters

other	The object with which to compare for inequality.

ccOCVMaxTrainParams

ccOCVMaxTuneParams

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxTuneParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The OCVMax tune parameters specify which aspects of each paragraph within an arrangement should be modified by tuning, and they also allow specifying a callback so that the user can track the progress of a tune operation, which can be time consuming.

Constructors/Destructors

ccOCVMaxTuneParams

```
ccOCVMaxTuneParams();

ccOCVMaxTuneParams(const ccOCVMaxArrangement& arr);
```

- `ccOCVMaxTuneParams();`

  Constructs a default tune parameters object.

- `ccOCVMaxTuneParams(const ccOCVMaxArrangement& arr);`

  Constructs a tune parameters object with one paragraph tune parameters object for each paragraph of the arrangement.

  Parameters

  arr The arrangement.

  Notes

  The default copy constructor, assignment operator, and destructor are used.

Public Member Functions

paragraphIndices

```
const cmStd vector<c_Int32>& paragraphIndices() const;

void paragraphIndices(const cmStd vector<c_Int32>& p);
```

- `const cmStd vector<c_Int32>& paragraphIndices() const;`

  Gets which paragraphs to tune.

- `void paragraphIndices(const cmStd vector<c_Int32>& p);`

  Sets which paragraphs to tune.

  An empty vector indicates that all paragraphs should be tuned.

  Parameters

  p The paragraphs to be tuned.

paragraphParams

```
const cmStd vector<ccOCVMaxParagraphTuneParams>
 &paragraphParams() const;

void paragraphParams(const cmStd
 vector<ccOCVMaxParagraphTuneParams> &pparams);
```

- `const cmStd vector<ccOCVMaxParagraphTuneParams> &paragraphParams() const;`

  Gets the parameters for each paragraph to be tuned.

- `void paragraphParams(const cmStd vector<ccOCVMaxParagraphTuneParams> &pparams);`

  Sets the parameters for each paragraph to be tuned.

  The vector is referenced using the indices specified by paragraphIndices. The vector should have the same number of elements as the tuned arrangement has paragraphs.

  Parameters

  pparams The paragraph parameters.

callback

```
const ccOCVMaxProgressCallback& callback() const;
void callback(const ccOCVMaxProgressCallback& c);
```

- `const ccOCVMaxProgressCallback& callback() const;`

  Gets a callback function for tuning.

- `void callback(const ccOCVMaxProgressCallback& c);`

  Sets a callback function for tuning.

  The callback function will be called at various points during the tool's **tune()** function to indicate the amount of progress made, to allow implementing functionality such as a progress bar. The callback will initially be called with a progress value of 0.0, and near the end of tuning it will be called with a value of 1.0; it may be called with several increasing values in between.

  Parameters

  c The callback function.

  Notes

  The callback object is not serialized.

ccOCVMaxTuneParams

ccOCVMaxTuneResult

```
#include <ch_cvl/ocvmax.h>

class ccOCVMaxTuneResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccOCVMaxTuneResult** class contains information obtained by tuning an OCVMax tool.

Constructors/Destructors

ccOCVMaxTuneResult

```
ccOCVMaxTuneResult();
```

Constructs a default tune result.

Notes

The default copy constructor, assignment operator, and destructor are used.

Public Member Functions

result

```
const ccOCVMaxResult& result() const;
```

Gets the result, which represents the run result obtained for the final tuned arrangement.

runParams

```
const ccOCVMaxRunParams &runParams() const;
```

Gets a *runParams* object which is a tightly restricted set of params that is sufficiently open to find the tuned arrangement in the tuning image.

runTimeout

```
double runTimeout() const;
```

Gets a timeout value which is sufficiently large to allow a run to complete.

ccOCVMaxTuneResult

startPose

```
const cc2XformBasePtrh_const& startPose() const;
```

Gets a start pose that is appropriate to find the tuned arrangement in the tuning image.

resetHistory

```
void resetHistory();
```

Resets the history so that previous tuning data is not used to tune the output run parameters.

Notes

This affects only **ccOCVMaxTool::tuneRunParams()**.

ccOCVPosResult

```
#include <ch_cvl/ocv.h>

class ccOCVPosResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class captures information about the OCV tool's verification results for a character position within a line of text. You should not create a **ccOCVPosResult** directly.

Constructors/Destructors

ccOCVPosResult

```
ccOCVPosResult();
```

Constructs a position result object with a verification status of failed.

Operators

operator==

```
bool operator==(const ccOCVPosResult& other) const;
```

Returns true if this object is equal to the specified position results object.

Parameters

other The position results object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVPosResult& other) const;
```

Returns true if this object is not equal to the specified position results object.

Parameters

other A position results object with which to compare for inequality.

Public Member Functions

posIndex

```
c_Int32 posIndex() const;
```

Returns the index of the character position within the line.

key

```
c_Int32 key() const;
```

Returns the key of the character found at the position. A value of -1 is returned if no character was found at this position or if a wildcard key is part of the current key set.

status

```
ccOCVDefs::CharStatus status() const;
```

Returns the verification status for this character position: verified, confused, or failed.

clientPose

```
const cc2Xform& clientPose() const;
```

Returns the pose of the character in client coordinates.

If no character was found at this position, a blank character was defined for this position, or a wildcard key is part of the current key set, then the returned pose is estimated from the found poses of adjacent characters.

linePose

```
const cc2Xform& linePose() const;
```

Returns the pose of the character in line coordinates.

If no character was found at this position, a blank character was defined for this position, or a wildcard key was part of the current key set, then the returned pose is estimated from the found poses of adjacent characters.

score

```
double score() const;
```

Returns the verification score, which indicates how well the character found at this position matched the expected character.

A value of 0.0 is returned if no character was found. If a wildcard key is part of the current key set, a value of 1.0 is returned. In all cases, the returned value is greater than the accept threshold if the character was found.

confidenceScore

```
double confidenceScore() const;
```

Returns the confidence score for the character. The confidence score indicates the amount that the verification score for this character is greater than the next-highest scoring character that has a non-zero confusion score with the character at the indexed position. Characters corresponding to the current key indices are not able to be confused with the position.

A value of 0.0 is returned if no character was found or the highest-scoring confusion character's score exceeded this character's score.

For blank characters, this score indicates the tool's confidence that the tool has that no features are present.

confusionKeys

```
const cmStd vector<c_Int32>& confusionKeys() const;
```

Returns the keys of all the characters that were confused at this position.

confusionMatchScores

```
const cmStd vector<double>& confusionMatchScores() const;
```

Returns the verification scores received by the characters that were confused at this position. The order of the scores matches the order of the keys returned by **confusionKeys()**.

ccOCVPosResult

ccOCVPosRunParams

```
#include <ch_cvl/ocv.h>

class ccOCVPosRunParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class defines the run-time parameters used by the OCV tool to verify a character position within a line of text.

Constructors/Destructors

ccOCVPosRunParams

```
ccOCVPosRunParams(c_Int32 posIndex = 0,
  double acceptThreshold = 0.5,
  double confidenceThreshold = 0.0);
```

Constructs a run-time parameters object for the specified position within a line.

Parameters

posIndex The index of the character position for which these run-time parameters apply.

acceptThreshold The matching score below which the character at this line position fails verification.

confidenceThreshold
The confidence threshold determines whether the character position receives a verification status of verified or confused. If the difference between the matching score for the position and the highest scoring confusing character is greater than the confidence threshold, the position passes verification; otherwise, the verification status of the position is confused.

Throws

ccOCVDefs::BadIndex
posIndex is less than 0.

ccOCVDefs::BadParams
acceptThreshold or *confidenceThreshold* is less than 0 or greater than 1.

Operators

operator==

```
bool operator==(const ccOCVPosRunParams& other) const;
```

Returns true if this object is equal to the specified object.

Parameters

other The run-time parameters object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVPosRunParams& other) const;
```

Returns true if this object is not equal to the specified object.

Parameters

other The run-time parameters object with which to compare for inequality.

Public Member Functions

posIndex

```
void posIndex(c_Int32 index);
c_Int32 posIndex() const;
```

- `void posIndex(c_Int32 index);`

Sets the character position index for these run-time parameters.

Parameters

index A position index.

Throws

ccOCVDefs::BadIndex
posIndex is less than 0.

- `c_Int32 posIndex() const;`

Returns the character position index for these run-time parameters.

acceptThreshold

```
void acceptThreshold(double t);
double acceptThreshold() const;
```

- `void acceptThreshold(double t);`

  Sets the accept threshold for this character position. The accept threshold specifies the matching score below which the character at this position fails optical verification.

  Parameters

  t The accept threshold. This value must be between 0 and 1.

  Throws

  ccOCVDefs::BadParams
  t is less than 0 or greater than 1.

- `double acceptThreshold() const;`

  Returns the accept threshold for this line position.

confidenceThreshold

```
void confidenceThreshold(double t);
double confidenceThreshold() const;
```

- `void confidenceThreshold(double t);`

  Sets the confidence threshold for the character position. The confidence threshold determines whether a character position receives a verification status of verified or confused. If the difference between the matching score for the position and the highest scoring confusing character is greater than the confidence threshold, the position passes verification; otherwise, the verification status of the position is confused.

  Parameters

  t The confidence threshold. This value must be between 0 and 1.

  Throws

  ccOCVDefs::BadParams
  t is less than 0 or greater than 1.

- `double confidenceThreshold() const;`

  Returns the confidence threshold for the character position.

ccOCVPosRunParams

ccOCVResult

```
#include <ch_cvl/ocv.h>

class ccOCVResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class captures the result information obtained by using the OCV tool to verify a line arrangement.

Constructors/Destructors

ccOCVResult

```
ccOCVResult();
```

Constructs a result object with a failed status and no line results.

Operators

operator==

```
bool operator==(const ccOCVResult& other) const;
```

Returns true if this object is equal to the specified result object.

Parameters

other	The OCV tool result object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVResult& other) const;
```

Returns true if this object is not equal to the specified result object.

Parameters

other	The OCV tool result object with which to compare for inequality.

Public Member Functions

clientPose

```
const cc2Xform& clientPose() const;
```

Returns the pose of the line arrangement in client coordinates.

fixturePose

```
const cc2Xform& fixturePose() const;
```

Returns the pose of the arrangement fixture in client coordinates.

score

```
double score() const;
```

Returns the overall verification score for the line arrangement. This score is the average of the verification scores for the lines that the tool attempted to verify.

numLinesVerified

```
c_Int32 numLinesVerified() const;
```

Returns the number of lines that passed verification.

verified

```
bool verified() const;
```

Returns true if all lines in the line arrangement passed verification. A line passes verification if all character positions in the line receive a status of verified.

lineResults

```
const cmStd vector<ccOCVLineResult>& lineResults() const;
```

Returns the results for all lines in the arrangement for which verification has been attempted.

lineResult

```
const ccOCVLineResult& lineResult(c_Int32 lineIndex) const;
```

Returns the result for the specified line.

Parameters

lineIndex The index of the line.

Throws

ccOCVDefs::BadIndex
lineIndex is not the index of a line for which verification was attempted.

time

```
double time() const;
```

Returns the time in milliseconds that the tool takes to complete its run.

ccOCVRunParams

```
#include <ch_cvl/ocv.h>

class ccOCVRunParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class encapsulates the parameters required to run the OCV tool.

Constructors/Destructors

ccOCVRunParams

```
ccOCVRunParams();

ccOCVRunParams (
  const cmStd vector<ccOCVLineRunParams>& lineParams,
  const cc2Xform& expectedPose = cc2Xform(),
  const cc2Rigid& fixtureOffset = cc2Rigid(),
  double translationUncertainty = 0,
  const ccRadian & rotationUncertainty = ccRadian(0),
  double scaleUncertainty = 0,
  double timeout = HUGE_VAL);

ccOCVRunParams (
  const ccOCLineArrangement& lineArrangement,
  const cc2Xform& expectedPose,
  const cc2Rigid& fixtureOffset,
  double translationUncertainty = 0,
  const ccRadian & rotationUncertainty = ccRadian(0),
  double scaleUncertainty = 0,
  double acceptThreshold = 0.5,
  double confidenceThreshold = 0.0,
  double timeout = HUGE_VAL);

ccOCVRunParams (
  const ccOCLineArrangement& lineArrangement,
  const cc2Xform& expectedPose = cc2Xform(),
  c_Int32 lineIndex = 0,
  c_Int32 posIndex = 0,
  double translationUncertainty = 0,
```

```
  const ccRadian & rotationUncertainty = ccRadian(0),
  double scaleUncertainty = 0,
  double acceptThreshold = 0.5,
  double confidenceThreshold = 0.0,
  double timeout = HUGE_VAL);
```

- `ccOCVRunParams();`

  Constructs a default run-time parameters object with an identity expected pose, no offset, and no uncertainty factors.

- ```
  ccOCVRunParams (
    const cmStd vector<ccOCVLineRunParams>& lineParams,
    const cc2Xform& expectedPose = cc2Xform(),
    const cc2Rigid& fixtureOffset = cc2Rigid(),
    double translationUncertainty = 0,
    const ccRadian & rotationUncertainty = ccRadian(0),
    double scaleUncertainty = 0,
    double timeout = HUGE_VAL);
  ```

  Constructs a run-time parameters object with the specified expected pose, offset, and uncertainties.

  Parameters

  lineParams The line run-time parameters for each of the lines in the arrangement to be verified.

  expectedPose The nominal pose of the fixture, specified in the client coordinates of the input image.

  fixtureOffset The rigid offset from arrangement coordinates to the fixture.

  translationUncertainty
  The translation uncertainty of the line arrangement.

  rotationUncertainty
  The rotation uncertainty of the line arrangement.

  scaleUncertainty
  The scale uncertainty of the line arrangement. You should not specify a value greater than 0.10 for the scale uncertainty.

  timeout The maximum time in seconds for the tool to run.

  Throws

  ccOCVDefs::BadParams
  translationUncertainty, *rotationUncertainty*, *scaleUncertainty,* or *timeout* is less than 0.
  ```

ccOCVDefs::BadIndex
lineParams contains repeated position indices.

- ```
  ccOCVRunParams (
    const ccOCLineArrangement& lineArrangement,
    const cc2Xform& expectedPose,
    const cc2Rigid& fixtureOffset,
    double translationUncertainty = 0,
    const ccRadian & rotationUncertainty = ccRadian(0),
    double scaleUncertainty = 0,
    double acceptThreshold = 0.5,
    double confidenceThreshold = 0.0,
    double timeout = HUGE_VAL);
  ```

  Constructs a run-time parameters object with the specified expected pose, offset, and uncertainties. The run-time parameters for each line and character position in the supplied arrangement are automatically constructed using the given accept and confidence thresholds.

  Parameters

  lineArrangement The line arrangement with lines to be verified.

  expectedPose The nominal pose of the fixture, specified in the client coordinates of the input image.

  fixtureOffset The rigid offset from arrangement coordinates to the fixture.

  translationUncertainty
  The translation uncertainty of the line arrangement.

  rotationUncertainty
  The rotation uncertainty of the line arrangement.

  scaleUncertainty
  The scale uncertainty of the line arrangement. You should not specify a value greater than 0.10 for the scale uncertainty.

  acceptThreshold The accept threshold, which specifies the matching score below which a character position fails verification. This value applies to all character positions in the line arrangement.

  confidenceThreshold
  The confidence threshold, which determines whether a character position receives a verification status of verified or confused. If the difference between the matching score for the position and the highest scoring confusing character is greater than the confidence threshold, the position passes verification; otherwise,
  ```

the verification status of the position is confused. The confidence threshold value applies to all character positions in the line arrangement.

timeout The maximum time in seconds for the tool to run.

Throws

ccOCVDefs::BadParams

translationUncertainty, *rotationUncertainty*, *scaleUncertainty*, or *timeout* is less than 0; or the *acceptThreshold* or *confidenceThreshold* is less than 0 or greater than 1.

Notes

When the tool runs, it verifies all character positions in all lines.

- ```
  ccOCVRunParams (
    const ccOCLineArrangement& lineArrangement,
    const cc2Xform& expectedPose = cc2Xform(),
    c_Int32 lineIndex = 0,
    c_Int32 posIndex = 0,
    double translationUncertainty = 0,
    const ccRadian & rotationUncertainty = ccRadian(0),
    double scaleUncertainty = 0,
    double acceptThreshold = 0.5,
    double confidenceThreshold = 0.0,
    double timeout = HUGE_VAL);
  ```

Constructs a run-time parameters object with the specified expected pose and uncertainties, and computes the offset from the fixture to the specified character position. Run-time parameters for line and character positions in the line arrangement are automatically constructed using the supplied accept and confidence thresholds.

Parameters

lineArrangement The line arrangement with lines to be verified.

expectedPose The nominal pose of the fixture, specified in the client coordinates of the input image.

lineIndex The index of the line that contains the character from which to calculate the fixture offset.

posIndex The index of the character position from which to calculate the fixture offset.

translationUncertainty

The translation uncertainty of the line arrangement.
  ```

rotationUncertainty
The rotation uncertainty of the line arrangement.

scaleUncertainty
The scale uncertainty of the line arrangement. You should not specify a value greater than 0.10 for the scale uncertainty.

acceptThreshold The accept threshold, which specifies the matching score below which a character position fails verification. This value applies to all character positions in the line arrangement.

confidenceThreshold
The confidence threshold, which determines whether a character position receives a verification status of verified or confused. If the difference between the matching score for the position and the highest scoring confusing character is greater than the confidence threshold, the position passes verification; otherwise, the verification status of the position is confused. The confidence threshold value applies to all character positions in the line arrangement.

timeout The maximum time in seconds for the tool to run.

Throws

ccOCVDefs::BadParams
translationUncertainty, *rotationUncertainty*, *scaleUncertainty*, or *timeout* is less than 0; or the *acceptThreshold* or *confidenceThreshold* is less than 0 or greater than 1.

ccOCVDefs::BadIndex
The *lineIndex* or *posIndex* value is not valid for the given arrangement.

Notes

When the tool runs, it verifies all characters in all lines.

Operators

operator==

```
bool operator==(const ccOCVRunParams& other) const;
```

Returns true if this object is equal to the specified run-time parameters object.

Parameters

other The object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVRunParams& other) const;
```

Returns true if this object is not equal to the specified run-time parameters object.

Parameters

other The object with which to compare for inequality.

Public Member Functions

expectedPose

```
void expectedPose(const cc2Xform& pose);

const cc2Xform& expectedPose() const;
```

- `void expectedPose(const cc2Xform& pose);`

  Sets the expected pose of the fixture, specified in the client coordinates of the input image.

  Parameters

  pose The expected pose of the arrangement fixture.

- `const cc2Xform& expectedPose() const;`

  Returns the expected pose of the fixture, specified in the client coordinates of the input image.

fixtureOffset

```
void fixtureOffset(const cc2Rigid& offset);

const cc2Rigid& fixtureOffset() const;

void fixtureOffset(const ccOCLineArrangement& lineArr,
  c_Int32 lineIndex, c_Int32 posIndex);
```

- `void fixtureOffset(const cc2Rigid& offset);`

  Sets the offset from the arrangement coordinates to the fixture.

  Parameters

  offset The offset from the line arrangement to the fixture.

- `const cc2Rigid& fixtureOffset() const;`

  Returns the offset from the arrangement coordinates to the fixture.

- `void fixtureOffset(const ccOCLineArrangement& lineArr, c_Int32 lineIndex, c_Int32 posIndex);`

  Sets the offset from the line arrangement to a fixture that is defined by the coordinate system of the specified character within the line arrangement.

  Parameters

  lineArr The line arrangement for which to set the fixture offset.

  lineIndex The index of the line that contains the character position to act as the fixture.

  posIndex The index of the character position to act as the fixture.

  Throws

  ccOCVDefs::BadIndex
  The *lineIndex* or *posIndex* value is not valid for the given arrangement.

translationUncertainty

```
void translationUncertainty(double unc);

double translationUncertainty() const;
```

- `void translationUncertainty(double unc);`

  Sets the translation uncertainty for the expected pose of the line arrangement.

  Parameters

  unc The translation uncertainty. This must be greater than or equal to 0.

  Throws

  ccOCVDefs::BadParams
  unc is less than 0.

- `double translationUncertainty() const;`

  Returns the translation uncertainty for the expected pose of the line arrangement.

rotationUncertainty

```
void rotationUncertainty(const ccRadian& unc);
const ccRadian& rotationUncertainty() const;
```

- `void rotationUncertainty(const ccRadian& unc);`

  Sets the rotation uncertainty for the expected pose of the line arrangement.

  Parameters

  unc The rotation uncertainty. This must be greater than or equal to 0.

  Throws

  ccOCVDefs::BadParams
  unc is less than 0.

- `const ccRadian& rotationUncertainty() const;`

  Returns the rotation uncertainty for the expected pose of the line arrangement.

scaleUncertainty

```
void scaleUncertainty(double unc);
double scaleUncertainty() const;
```

- `void scaleUncertainty(double unc);`

  Sets the scale uncertainty for the expected pose of the line arrangement.

  Parameters

  unc The scale uncertainty. This must be greater than or equal to 0. You should not specify a value greater than 0.10 for the scale uncertainty.

  Throws

  ccOCVDefs::BadParams
  unc is less than 0.

- `double scaleUncertainty() const;`

  Returns the scale uncertainty for the expected pose of the line arrangement.

lineParams

```
void lineParams(
  const cmStd vector<ccOCVLineRunParams>& params);

const cmStd vector<ccOCVLineRunParams>& lineParams() const;
```

- ```
  void lineParams(
    const cmStd vector<ccOCVLineRunParams>& params);
  ```

  Sets the run-time parameters for the lines to be verified.

  Parameters

  params The line run-time parameters.

  Throws

  ccOCVDefs::BadIndex

  params contains repeated line indices.

- ```
  const cmStd vector<ccOCVLineRunParams>& lineParams() const;
  ```

  Returns the run-time parameters for the lines to be verified.

timeout

```
void timeout(double timeInSeconds);

double timeout() const;
```

- ```
  void timeout(double timeInSeconds);
  ```

  Sets the maximum time in seconds allowed for the tool to run.

  Parameters

  timeInSeconds The maximum time in seconds for the tool's run.

  Throws

  ccOCVDefs::BadParams

  timeInSeconds is less than 0.

  ccTimeout::Expired

  The tool's **run()** function has not completed before the timeout period elapses. As a consequence, the tool does not yield valid results.
  ```

Notes

This value is intended as an approximate upper limit. The actual execution time may exceed the maximum timeout value by up to 0.02 seconds on a 200 Mhz Pentium, however typical latencies for faster CPU machines are much lower.

User-created **ccTimeout** objects can also terminate the tool's **run()** function.

If you did not specify a timeout for the run-time parameters object at construction-time, it defaults to *HUGE_VAL*.

- `double timeout() const;`

Returns the maximum time in seconds allowed for the tool to run.

ccOCVTool

```
#include <ch_cvl/ocv.h>

class ccOCVTool;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class contains methods for training and running the OCV tool. You use the OCV tool to perform optical character verification of a text area.

Constructors/Destructors

ccOCVTool

```
ccOCVTool();

~ccOCVTool();

ccOCVTool(const ccOCVTool& t);
```

- `ccOCVTool();`

  Constructs an untrained OCV tool.

- `~ccOCVTool();`

  Destroys the tool.

- `ccOCVTool(const ccOCVTool& t);`

  Creates a copy of the specified OCV tool.

  Parameters

t	The tool to copy.

operator=

```
ccOCVTool& operator=(const ccOCVTool& t);
```

Assigns the values of the specified OCV tool to this tool.

Parameters

t The tool from which to assign values.

Operators

operator==

```
bool operator==(const ccOCVTool& other) const;
```

Returns true if this object is equal to the specified tool object.

Parameters

other The OCV tool object with which to compare for equality.

operator!=

```
bool operator!=(const ccOCVTool& other) const;
```

Returns true if this object is not equal to the specified tool object.

Parameters

other The OCV tool object with which to compare for inequality.

Public Member Functions

train

```
void train(
  const ccOCLineArrangementPtrh_const& lineArrangement,
  ccDiagObject* diagObject = 0,
  c_UInt32 diagFlags = 0);
```

Trains the tool using the supplied line arrangement.

If alphabets referenced by the lines in the line arrangement have not yet been compiled, they are automatically compiled as a result of calling **train()**. This may cause the training operation to take substantially longer and may also affect other clients of the alphabets.

Parameters

lineArrangement The line arrangement with which to train the tool.

diagObject A valid diagnostic object or null.

diagFlags Diagnostic flags.

Throws

ccOCVDefs::CanNotTrain

The arrangement contains no lines; or any line in the arrangement has wildcard or blank characters in all positions.

retrain

```
void retrain(
  const ccOCLineArrangementPtrh_const& lineArrangement,
  ccDiagObject* diagObject = 0,
  c_UInt32 diagFlags = 0);
```

Retrains the tool using the specified line arrangement. Only tool information that has changed since the last training operation is retrained.

Parameters

lineArrangement The line arrangement with which to retrain the tool.

diagObject A valid diagnostic object or null.

diagFlags Diagnostic flags.

Throws

ccOCVDefs::CanNotTrain
The arrangement contains no characters.

ccOCVDefs::NotTrained
The tool is not trained.

untrain

```
void untrain();
```

Untrains the tool, releasing any memory that was allotted by training.

Notes

Calling this function has no effect if the tool is not trained.

isTrained

```
bool isTrained() const;
```

Returns true if this tool is trained.

lineArrangement

```
const ccOCLineArrangementPtrh_const&
  lineArrangement() const;
```

Returns the trained line arrangement to be verified by the tool.

Throws

ccOCVDefs::NotTrained
The tool is not trained.

run

```
void run(const ccPelBuffer_const<c_UInt8> image,
  const ccOCVRunParams& runParams,
  ccOCVResult& results,
  ccDiagObject* diagObject = 0,
  c_UInt32 diagFlags = 0) const;
```

Runs the OCV tool using the supplied image and runtime parameters, and records tool results in the supplied result object.

The tool tries to verify all characters specified by its run-time parameters. However, following a verification failure, the tool may not be able to locate remaining characters. In this case, the tool reports the results of those characters verified up to and including the verification failure.

Parameters

image The image that contains the text area to verify.

runParams Parameters for the tool that specify the run-time arrangement of lines and characters and the pose and uncertainties of these.

results The tool stores information about the verification run in the *results* object.

diagObject A valid diagnostic object or null.

diagFlags Any diagnostic flags.

Throws

ccOCVDefs::NotTrained
tool is not currently trained.

ccOCVDefs::BadImage
The image is unbound.

ccOCVDefs::BadIndex
The run-time parameters specify a line or character position index that does not correspond to a valid line or character position in the supplied arrangement.

ccTimeout::Expired
The tool timeout period elapsed.

draw

```
void draw(const ccOCVResult& result, c_UInt32 drawFlags,
  ccGraphicList& graphList) const;
```

Appends the specified graphics associated with the supplied result to the supplied graphics list. The graphics are colored as follows: green for passed, red for failed, blue for confused, and yellow for not run.

Notes

The supplied result must have been generated by calling **run()** while the tool was in its current trained state.

Parameters

result	The result for which to create graphics.
drawFlags	A value formed by ORing together zero or more of the values defined in **ccOCVDefs::DrawFlags**.

Throws

ccOCVDefs::NotTrained
This tool is not trained.

ccOCVTool

ccOutputLine

```
#include <ch_cvl/pio.h>

class ccOutputLine;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

ccOutputLine describes a single output line. You obtain a **ccOutputLine** by calling the **outputLine()** member of the class that describes your hardware platform.

You can set any output line to **true** or **false** using **ccOutputLine::set**. For TTL output or TTL bidirectional lines, **true** means TTL high (+5V). For opto-isolated lines, **true** enables the no-current-flowing state (the LED inside the opto-isolator is OFF, and thus the circuit is open). For opto-isolated lines, **false** sets the current-flowing state (the LED inside the opto-isolator is ON, thus the circuit is closed). Refer to the circuit diagram for opto-isolated parallel I/O lines in the hardware manual for your MVS-8000 product.

Notes

Do not attempt to construct a **ccOutputLine** directly.

Public Member Functions

get

```
bool get() const;
```

Retrieves the current state of the output line associated with this **ccOutputLine**. The meanings of **true** and **false** are defined in *Class Properties* above.

This function does not indicate the actual state of the line. Instead, it indicates the state that the software is attempting to set. An improper connection or hardware failure can cause the actual state of the output line to be different than the value returned by this function.

Throws

ccParallelIO::NotEnabled

The output line associated with this **ccOutputLine** is not enabled. You must enable the output line by calling **enable()** before you can examine its state.

set `void set(bool value);`

Sets the output line associated with this **ccOutputLine** to the specified state.

Parameters

value Set *value* to **true** to set the output line to TTL high (for TTL output or bidirectional lines) or to the no-current-flowing state (for opto-isolated lines). Set *value* to **false** to set the output line to TTL low (for TTL lines) or to current-flowing state (for opto-isolated lines).

Throws

ccParallelIO::NotEnabled
The output line associated with this **ccOutputLine** is not enabled. You must enable the output line by calling **enable()** before you can set its state.

lineNumber `c_Int32 lineNumber() const;`

Returns the logical line number of the output line associated with this **ccOutputLine**. The returned value is in the range determined by the configuration set by **ccParallelIO::setIOConfig()**.

Throws

ccParallelIO::NotEnabled
The output line object was not constructed correctly.

Notes

See the documentation for the class that describes your hardware platform for the equivalence between logical line numbers and signal names.

enable `void enable(bool value);`

Enables or disables the output line associated with this **ccOutputLine**.

Parameters

value Set *value* to **true** to enable the line, **false** to disable it.

Throws

ccParallelIO::CannotEnable
This **ccOutputLine** cannot be enabled. On some hardware platforms, all bidirectional TTL lines must be enabled as a group to be input or output lines. In these cases, it may not be possible to enable an output line if other lines in the same group have been configured as input lines.

enabled

```
bool enabled() const;
```

Returns **true** if the output line associated with this is **ccOutputLine** enabled and **false** otherwise.

Throws

ccParallelIO::NotEnabled
The output line object was not constructed correctly.

canEnable

```
bool canEnable() const;
```

Returns **true** if the output line associated with this **ccOutputLine** can be enabled. Returns **false** if **enable()** would throw an error.

Throws

ccParallelIO::NotEnabled
The output line object was not constructed correctly.

toggle

```
void toggle();
```

Sets the line to the opposite of its current state.

Throws

ccParallelIO::NotEnabled
The output line associated with this **ccOutputLine** is not enabled. You must enable the output line by calling **enable()** before you can set its state.

pulse

```
void pulse(bool polarity, double width,
  bool blocking=true);
```

Toggles the line to the specified polarity for the specified number of seconds.

Parameters

polarity — True means a zero-to-one transition; false means a one-to-zero transition.

width — The number of seconds to wait before complementing the line.

blocking — Specifies how you want this function to behave. True makes it a blocking function, false makes it a non-blocking function.

Throws

ccParallelIO::NotEnabled
The output line associated with this **ccOutputLine** is not enabled. You must enable the output line by calling **enable()** before you can set its state.

Notes

1. If the pulse is disabled or the program exits in the middle of a pulse, the line retains its last state.
2. If the polarity argument specified is the same as the current state, the pulse function does not toggle on the initial transition. For example, if the state of the output line is already high (one) and the user attempts to pulse high (*polarity* = **true**), the line will remain high on what would have been the initial pulse transition, and after the pulse duration expires, it will toggle low.
3. The first call that your program makes to this function may result in a pulse that is from ten to several hundred milliseconds longer than you specified. This additional time only occurs the first time you call **pulse()**; subsequent pulses will be the correct length, subject to the granularity limits described above.
4. The **pulse()** function has a granularity of 1 ms for embedded applications running on the MVS-82400. For frame grabber-based applications running on a PC host, the pulse granularity is approximately 10 ms, although the exact timing is controlled by the Windows OS. In this case, you can improve the resolution of the pulse by changing the granularity of the operating system's multimedia timer, using the Win32 function **timeBeginPeriod()**.

   However, be aware that changing the operating system's timer resolution affects the timing of all programs running on that system. Changing the resolution may adversely affect the timing of CVL image acquisition or vision tool operations. Carefully test all aspects of your CVL application if you change the operating system timer resolution.
5. For PC-based frame grabber applications, the actual pulse width can be longer than the width you specify depending on the host PC processor speed, the amount of memory, and the operating environment when the **pulse()** call is made. For example, the number of background tasks, amount of background activity, and so on can increase the pulse width beyond what you specify.

   When using non-blocking pulses with PC-based frame grabber applications, the actual pulse width can be slightly shorter than that specified due to known limitations with Windows OS timers. The problem is exacerbated by making multiple non-blocking **pulse()** calls on multiple lines simultaneously. If you need an absolute minimum pulse width, increase the specified pulse width a small amount. Please note, however, that this will increase the average width for all pulses. This problem does not occur for blocking **pulse()** calls.
6. Calling the **pulse()** function again replaces any pulse in progress on the same physical line.

ccOverrunCallbackProp

```
#include <ch_cvl/prop.h>

class ccOverrunCallbackProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

As of CVL 6.1, this class is one of two ways to register a callback class for the overrun state:

Method	Features	See
ccAcqFifo:: overrunInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	*overrunInfoCallback* on page 234
ccOverrunCallbackProp	Calls your callback function.	this section

If you are writing a new CVL application, consider using **ccAcqFifo::overrunInfoCallback()** to register your move-part callback function.

This class describes the overrun callback property of an acquisition FIFO queue. The class contains a pointer to a function you write that is called by the acquisition software when an overrun occurs. The acquisition software calls the overrun callback function when an overrun occurs.

The callback function you write should set flags or semaphores in your application that allow your program to handle the overrun condition the next time it executes. Your callback function is executed internally by the acquisition software, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccOverrunCallbackProp

```
ccOverrunCallbackProp();

explicit ccOverrunCallbackProp(
  const ccCallbackPtrh &callback);
```

- `ccOverrunCallbackProp();`

  Creates a new overrun callback property not associated with any FIFO. The default callback function is an unbound handle, meaning no callback function is invoked.

- `explicit ccOverrunCallbackProp(`
  `const ccCallbackPtrh &callback);`

  Creates a new overrun callback function not associated with any FIFO, registering the function pointed to by *callback* as the overrun callback function.

  Parameters

  callback — Effectively, the callback function to be invoked when the acquisition process detects an overrun error or an **isMissed()** error. Generally, overrun errors and **isMissed()** errors occur when triggers are occurring too fast. See **ccAcqFailure::isOverrun()** and **ccAcqFailure::isMissed()** for more information. An unbound handle means no callback will occur. Technically, *callback* is a pointer handle to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

Public Member Functions

overrunCallback

```
void overrunCallback(const ccCallbackPtrh& callback);

const ccCallbackPtrh& overrunCallback() const;
```

- `void overrunCallback(const ccCallbackPtrh& callback);`

  callback points to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

Parameters

callback — Effectively, the callback function to be invoked when the acquisition process detects an overrun error or an **isMissed()** error. Generally, overrun errors and **isMissed()** errors occur when triggers are occurring too fast. See **ccAcqFailure::isOverrun()** and **ccAcqFailure::isMissed()** for more information. An unbound handle means no callback will occur. Technically, *callback* is a pointer handle to an instance of **ccCallback** that contains the callback function you have defined as an override of **operator()()**.

- `const ccCallbackPtrh& overrunCallback() const;`

  Returns the callback function to be invoked when the acquisition overruns.

ccOverrunCallbackProp

ccPackedRGB16Pel

```
#include <ch_cvl/colorpel.h>

class cmImport_cogstd ccPackedRGB16Pel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **ccPackedRGB16Pel** class describes an RGB pixel whose values are packed in a single 16-bit word. This format is also called 565 because of the way the values for each component are packed into 16 bits:

```
rrrr rggg gggb bbbb
```

Constructors/Destructors

ccPackedRGB16Pel

```
ccPackedRGB16Pel(c_UInt16 ppel=0);

ccPackedRGB16Pel(c_UInt8 pr, c_UInt8 pg, c_UInt8 pb);
```

- `ccPackedRGB16Pel(c_UInt16 ppel=0);`

  Creates a new 16-bit RGB pixel with the specified value.

  Parameters

  ppel — The value of the pixel. The default is 0 or black.

- `ccPackedRGB16Pel(c_UInt8 pr, c_UInt8 pg, c_UInt8 pb);`

  Creates a new 16-bit RGB pixel with the specified values for each of the RGB components.

  Parameters

  pr — The value for the red component. Must be a value from 0 through 31.

  pg — The value for the green component. Must be a value from 0 through 63.

pb The value of the blue component. Must be a value from 0 through 33.

Notes

Values outside the appropriate range are clipped to zero.

Public Member Functions

r

```
c_UInt8 r() const;
void r(c_UInt8 rv);
```

- `c_UInt8 r() const;`

Returns the red component of the pixel.

- `void r(c_UInt8 rv);`

Sets the red component of the pixel.

Parameters

rv The value for the red component. Must be a value from 0 through 31.

g

```
c_UInt8 g() const;
void g(c_UInt8 gv);
```

- `c_UInt8 g() const;`

Returns the green component of the pixel.

- `void g(c_UInt8 gv);`

Sets the green component of the pixel.

Parameters

gv The value for the red component. Must be a value from 0 through 63.

b

```
c_UInt8 b() const;
void b(c_UInt8 bv);
```

- `c_UInt8 b() const;`

  Returns the blue component of the pixel.

- `void b(c_UInt8 bv);`

  Sets the blue component of the pixel.

  Parameters

bv	The value for the red component. Must be a value from 0 through 31.

Operators

operator const c_UInt16&

```
operator const c_UInt16& () const;
```

Casts this pixel to a **const c_UInt16**.

operator c_UInt16&

```
operator c_UInt16& ();
```

Casts this pixel to a non-const **c_UInt16**.

operator==

```
bool operator==(const ccPackedRGB16Pel& p) const;
```

Compares this pixel to the specified pixel. Returns true if the two have the same RGB value, false otherwise.

Parameters

p	Pixel to compare to the current one.

operator!=

```
bool operator==(const ccPackedRGB16Pel& p) const;
```

Compares this pixel to the specified pixel. Returns true if the two have the same RGB value, false otherwise.

Parameters

p	Pixel to compare to the current one.

ccPackedRGB16Pel

ccPackedRGB32Pel

```
#include <ch_cvl/colorpel.h>

class cmImport_cogstd ccPackedRGB32Pel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **ccPackedRGB32Pel** class describes an RGB pixels whose values are packed into four 8-bit bytes. The three low order bytes contain the red, green, and blue information. The remaining byte contains an alpha value. When you acquire a 32-bit image through a 32-bit acquisition FIFO, you can specify how the alpha value is calculated with the FIFO's **ccAlphaColorProp** property.

Constructors/Destructors

ccPackedRGB32Pel

```
ccPackedRGB32Pel(c_UInt32 ppel=0);

ccPackedRGB32Pel(c_UInt8 pr, c_UInt8 pg, c_UInt8 pb,
  c_UInt8 pa);
```

- `ccPackedRGB32Pel(c_UInt32 ppel=0);`

  Creates a new 32-bit RGB pixel with the specified value.

  Parameters

ppel	The value of the pixel. The default is 0 or black.

- `ccPackedRGB32Pel(c_UInt8 pr, c_UInt8 pg, c_UInt8 pb, c_UInt8 pa);`

  Creates a new 32-bit RGB pixel with the specified values for each of the RGB components. Each value must be from 0 to 255.

  Parameters

pr	The value for the red component.
pg	The value for the green component.

pb The value of the blue component.

pa The value of the alpha component. This value is not used.

Public Member Functions

a

```
c_UInt8 a() const;
void a(c_UInt8 aval);
```

- `c_UInt8 a() const;`

  Returns the value of the alpha component.

- `void a(c_UInt8 aval);`

  Sets the value of the alpha component.

  Parameters

  aval The value of the alpha component.

r

```
c_UInt8 r() const;
void r(c_UInt8 rval);
```

- `c_UInt8 r() const;`

  Returns the value of the red component.

- `void r(c_UInt8 rval);`

  Sets the value of the red component.

  Parameters

  rval The value of the red component.

g

```
c_UInt8 g() const;
void g(c_UInt8 gval);
```

- `c_UInt8 g() const;`

  Gets the value of the green component.

- `void g(c_UInt8 gval);`

  Sets the value of the green component.

  Parameters

  gval The value of the green component.

b

`c_UInt8 b() const;`

`void b(c_UInt8 bval);`

- `c_UInt8 b() const;`

  Gets the value of the blue component.

- `void b(c_UInt8 bval);`

  Sets the value of the blue component.

  Parameters

  bval The value of the blue component.

Operators

operator const c_UInt32&

`operator const c_UInt32& () const;`

Casts this pixel to a **const c_UInt32**.

operator c_UInt32&

`operator c_UInt32& ();`

Casts this pixel to a non-const **c_UInt32**.

operator== `bool operator==(const ccPackedRGB32Pel& p) const;`

Compares this pixel to the specified pixel. Returns true if the two have the same values for their RGB and alpha components, false otherwise.

Parameters

p Pixel to compare to the current one.

operator!= `bool operator==(const ccPackedRGB32Pel& p) const;`

Compares this pixel to the specified pixel. Returns true if the two have the same values for their RGB and alpha components, false otherwise.

Parameters

p	Pixel to compare to the current one.

ccPair

```
#include <ch_cvl/pair.h>

template <class T> class ccPair : public cmStd pair<T,T>;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	yes

This class, derived from the standard template library's **pair** class, is used to describe pairs and arithmetic operations on the pairs. However, unlike the standard template library's class, both components of the pair must be of the same type. In the CVL, **ccPair** is often used to describe points.

T Template parameter specifying the type of the components. *T* must provide the following.

T()
T(const T&)
~T()
T& operator= (const T&)
operator+ (T, T)
operator+= (T)
operator- (T)
operator- (T, T)
operator-= (T)
operator* (T, T)
operator*= (T)
operator/ (T, T)
operator/= (T)
operator== (T, T)
operator!= (T, T)
T(c_Int16)
T(c_Int32)
T(float)
T(double)

Note that all builtin arithmetic types support these operations.

Typedefs are provided for **ccPair** for **c_Int32**, **c_Int16**, **double**, and **float**. See page 2361.

Constructors/Destructors

ccPair

```
ccPair();
ccPair(T x, T y);
```

- `ccPair();`

  Creates a new pair with components of type *T*. Both components are set to zero.

- `ccPair(T x, T y);`

  Crates a new pair with components of type *T* with initial values *x* and *y*.

  Parameters

x	Value of the x component.
y	Value of the y component.

Operators

operator+

```
ccPair<T> operator+ (const ccPair<T>& r) const;
```

Returns a new pair whose *x* and *y* components are added to the *x* and y components of *r*.

Parameters

r	The pair whose *x* and *y* values will be used in the addition.

operator+=

```
ccPair<T>& operator+= (const ccPair<T>& r);
```

Returns this pair after adding the *x* and *y* components or *r*.

Parameters

r	The pair whose *x* and *y* values will be used in the addition.

operator-

```
ccPair<T> operator- () const;
ccPair<T> operator- (const ccPair<T>& r) const;
```

- `ccPair<T> operator- () const;`

  Returns a new pair with the values of the *x* and *y* components negated. (Unary minus.)

- `ccPair<T> operator- (const ccPair<T>& r) const;`

  Returns a new pair with the x and y components of r subtracted from this pair's x and y components.

  Parameters

  r The pair whose *x* and *y* values will be used in the subtraction.

operator-= `ccPair<T>& operator-= (const ccPair<T>& r);`

Returns this pair after subtracting the *x* and *y* components of *r*.

Parameters

r The pair whose *x* and *y* values will be used in the subtraction.

operator* `ccPair<T> operator* (const ccPair<T>& r) const;`

Returns a new pair whose *x* and *y* components are multiplied by the *x* and y components of *r*.

Parameters

r The pair whose *x* and *y* values will be used in the multiplication.

operator*= `ccPair<T>& operator*= (const ccPair<T>& r);`

Returns this pair after multiplying the *x* and *y* components by the *x* and *y* components of *r*.

Parameters

r The pair whose *x* and *y* values will be used in the multiplication.

operator/ `ccPair<T> operator/ (const ccPair<T>& r);`

Returns a new pair with the *x* and *y* components divided by the *x* and *y* components of *r*.

r The pair whose *x* and *y* values will be used in the division.

operator/= `ccPair<T>& operator/= (const ccPair<T>& r);`

Returns this pair after dividing the *x* and *y* components by the *x* and *y* components of *r*.

Parameters

r The pair whose *x* and *y* values will be used in the division.

operator== `bool operator== (const ccPair<T>& r) const;`

Returns true if this pair and *r* are equal.

Parameters

r The pair to use in the comparison.

operator!= `bool operator!= (const ccPair<T>& r) const;`

Returns true if this pair and *r* are not equal.

Parameters

r The pair to use in the comparison.

Public Member Functions

x

```
const T& x() const;
T& x();
```

- `const T& x() const;`

  Returns a constant reference to the *x* component of the pair.

- `T& x();`

  Returns a reference to the *x* component of the pair.

y

```
const T& y() const;
T& y();
```

- `const T& y() const;`

  Returns a constant reference to the *y* component of the pair.

- `T& y();`

  Returns a reference to the *x* component of the pair.

Typedefs

ccSPair `typedef ccPair<c_Int16> ccSPair;`

ccIPair `typedef ccPair<c_Int32> ccIPair;`

ccFPair `typedef ccPair<float> ccFPair;`

ccDPair `typedef ccPair<double> ccDPair;`

ccPair

ccParallelIO

```
#include <ch_cvl/pio.h>

class ccParallelIO;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only.
Archiveable	No

ccParallelIO is an abstract base class for parallel input and output (I/O) from which are derived classes to represent specific Cognex hardware platforms.

When used with any member of the MVS-8100 frame grabber family, all member functions of **ccParallelIO** can throw *ccBoard::FpgaLoadFailure* on first use, if the external camera power cable is not connected to the frame grabber.

Constructors/Destructors

Constructors are automatically created for this class.

Public Member Functions

inputLine

```
virtual ccInputLine inputLine(c_Int32 line) = 0;
```

Returns a **ccInputLine** object that represents the specified logical input line.

Parameters

line The logical line number. *line* must be a valid value for the configuration set by **ccParallelIO::setIOConfig()**.

Throws

ccParallelIO::BadParams
line is not a valid value for the configuration set by **ccParallelIO::setIOConfig()**.

ccParalleIIO::NotImplemented
Input lines are not supported on the host side.

Notes

See the documentation for the derived class that describes your specific hardware platform for the equivalence between logical line numbers and physical lines.

outputLine `virtual ccOutputLine outputLine(c_Int32 line) = 0;`

Returns a **ccOutputLine** object that represents the specified logical output line.

Parameters

line The logical line. *line* must be a valid value for the configuration set by **ccParallelIO::setIOConfig()**.

Throws

ccParallelIO::BadParams
line is not a valid value for the hardware and the configuration set by **ccParallelIO::setIOConfig()**.

ccParalleIIO::NotImplemented
Output lines are not supported on the host side.

Notes

See the documentation for the derived class that describes your specific hardware platform for the equivalence between logical line numbers and physical lines.

numInputLines `virtual c_Int32 numInputLines() = 0;`

Returns the number of total input lines for this hardware. The total depends upon the configuration set by **ccParallelIO::setIOConfig()**.

Notes

Some platforms have bidirectional input/output lines. Refer to your hardware documentation for additional information on using bidirectional lines.

numOutputLines `virtual c_Int32 numOutputLines() = 0;`

Returns the number of total output lines for this hardware.

Notes

Some platforms have bidirectional input/output lines. Refer to your hardware documentation for additional information on using bidirectional lines.

getIOConfig `virtual ccIOConfig& getIOConfig()= 0;`

Returns the parallel I/O board configuration.

setIOConfig `virtual void setIOConfig(ccIOConfig& config)= 0;`

Sets the parallel I/O board configuration. See **ccIOConfig** and related classes in *ch_cvl/pioconfg.h*.

Parameters

config The I/O board configuration.

Example

This example sets the I/O configuration for an MVS-8120 frame grabber with a Standard Option Parallel I/O board. See **cc8120** for information about Parallel I/O board options.

```
cc8120& fg = cc8120::get(0);
ccParallelIO* pio = dynamic_cast<ccParallelIO*> (&fg);
pio-> setIOConfig(ccIOStandardOption());
```

Global Exceptions

A number of hardware-related global exceptions are defined as nested classes of **ccBoard**. These exceptions can be thrown by any member function in this class, or in classes derived from this class. These global exceptions include the following.

Throws

ccBoard::HardwareNotResponding

The frame grabber did not respond to the current access request. This can be the result of a problem with the hardware or the hardware's driver. Check that the board is installed and powered on correctly according to its hardware manual. Make sure there are no overcurrent conditions on parallel I/O lines. Check that the board's driver is running; check the Windows Event Log for any messages from the device driver.

ccBoard::HardwareInUse

The current process tried to access frame grabber hardware that is already owned by another running process. To avoid this error, a process that touches the hardware (such as a CVM ID query, number of camera ports query, or image acquisition request) must exit before another process can access the same hardware.

ccBoard::HardwareNotInitialized

The current access request received a response from the board's driver, but the board reports itself as not yet initialized. Make sure the current process has instantiated the right frame grabber class (**cc8100m**, **cc8504**, and so on). Power the host PC all the way off and back on and try the request again.

ccBoard::BadEERAMContents

The EERAM chip on the board that contains the board's serial number and other information could not be read.

ccBoard::FpgaLoadFailure

An error occured while loading the FPGA on the board. On some frame grabbers, including the MVS-8100M and MVS-8100C, this error can occur if external camera power is incorrectly applied to the board. Check the setting of the jumper that determines whether camera power is to be pulled from the PCI bus or from an external power cable, as described in the frame grabber's hardware manual. Make sure the external power cable, if used, is plugged into the board and the PC's power supply.

ccPDF417Result

```
#include <ch_cvl/pdf417.h>

class ccPDF417Result;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccPDF417Result** class contains a single result returned by the PDF417 decoder. It has const accessors for all available data.

Constructors/Destructors

ccPDF417Result

```
ccPDF417Result();
```

Constructs a **ccPDF417Result** with the following default values:

Parameter	Value
isDecoded_	False
decodedString_	“”
time_	0.0
rows_	-1
cols_	-1
numErrors_	-1
unusedErrorFraction_	-1

Public Member Functions

isDecoded

```
bool isDecoded () const;
```

Returns true if the result was both found and decoded, false otherwise. A result is decoded if the information encoded in the symbol under consideration was found and successfully decoded. To ensure a secure and valid decode, the return value of **ccPDF417Result::unusedErrorFraction()** should be a nonnegative value.

decodedString

```
ccCvlString decodedString () const;
```

Returns the decoded string.

Throws

ccPDF417Defs::BadString
The isDecoded() operation returned false.

time

```
double time () const;
```

Returns the time (in seconds) required to decode the symbol.

rows

```
c_Int32 rows () const;
```

Returns the number of data rows in the data region. A row is a lateral set of elements made up of a start pattern, codewords, and a stop pattern. The returned value ranges from 3 to 90. Returns -1 if this information is not available.

cols

```
c_Int32 cols () const;
```

Returns the number of data columns in the data region. In each row of a PDF417 symbol, between left and right row indicators, there may be from 1 to 30 data codewords. Collectively, among all rows, these codewords form data columns. The returned value ranges from 1 to 30. Returns -1 if this information is not available

numErrors

```
c_Int32 numErrors () const;
```

Returns the number of errors (both rejection and substitution) in data words encountered and corrected while decoding the symbol. Returns -1 if this information is not available.

unusedErrorFraction

```
double unusedErrorFraction () const;
```

Returns the ratio of the unused error correction codewords to the total number of error correction codewords specified by the error correction level encoded in the symbol. Among the total number of error correction codewords, two are needed for error detection. Any additional error correction codewords provide error correction capability, with one such codeword needed to recover each erasure (rejection) and two needed to correct each error (substitution).

Notes

The return value is in the range -1.0 through 1.0, inclusive. When the return value is -1.0, the decode procedure failed. When the return value is other negative values, the number of erasures and errors is beyond the limit for a safe correction. The closer a negative value is to 0.0, the lower the probability of a misread becomes.

In applications where a high decode rate is desired at the risk of low-probability misreads, you can use **ccPDF417Result::isDecoded()** as the sole indication of a successful decode. When a slightly lower but more secure decode rate is preferred to a high decode rate with possible misreads, check that the return value is nonnegative before considering the decode successful and valid.

operator==

```
bool operator== (const ccPDF417Result&) const;
```

Returns true if the ccPDF417 result passed in is equal to the current one, false otherwise.

Parameters

ccPDF417Result

PDF417 result object to compare.

Notes

Two ccPDF417Result objects are considered equal if all their corresponding data members except time_ are identical.

ccPDF417Result

ccPelBuffer

```
#include <ch_cvl/pelbuf.h>

template <class P> class ccPelBuffer :
  public ccPelBuffer_const<P>;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccPelBuffer** class is a template class for creating rectangular windows into root images with pixels of type *P* (**ccPelRoot<P>**).This class allows read/write access to the underlying pixels. The class **ccPelBuffer_const** allows read-only access to the pixels. Any number of windows may refer to the same root image.

P — Template parameter specifying the value associated with each pixel. CVL provides instantiations of **ccPelRoot** for the types **c_UInt8**, **c_UInt16**, **c_UInt32**, **ccPackedRGB16Pel**, and **ccPackedRGB32Pel**.

Functions that take pointers of type **P** as parameters or that return pointers of type **P** use the **pointer** or **const_pointer typedef** defined by **ccPelTraits** instead of **P***. See **ccPelTraits** on page 2403.

Constructors/Destructors

ccPelBuffer

```
ccPelBuffer();

ccPelBuffer(c_Int32 width, c_Int32 height,
  c_Int32 alignModulus=32);

ccPelBuffer(ccPelRoot<P>* pelroot);
```

- `ccPelBuffer();`

  Creates an unbound window, that is, one that is not associated with any root image. Height and width are set to 0 and all offsets are set to (0,0). The window's transform is set to identity.

- `ccPelBuffer(c_Int32 width, c_Int32 height, c_Int32 alignModulus=32);`

  Allocates contiguous storage for a root image of *width* x *height* pixels of type *P* and a window that encompasses the entire root image. The pixels are default-constructed. This storage will be freed by the destructor. The byte address of the first row's first pixel is zero modulo the *alignModulus*. The allocated row width (in pixels) is increased if necessary to make the row size (in bytes) a multiple of the specified *alignModulus*. Up to (*alignModulus*-1) pad pixels may be allocated for each row. Pad pixels, if any, occur to the right (greater x) of the pixels that belong to the root image. Pixel processing routines are permitted to overwrite pad pixels.

  Parameters

width	The width, in pixels, of the root image. (*width* > 0)
height	The height, in rows, of the root image. (*height* > 0)
alignModulus	A value that determines how memory used for the image is aligned. A value of 1 means that memory is aligned on byte boundaries, 16 aligns on word boundaries, 32 aligns on long word boundaries, and so on. (*alignModulus* >= 1)

- `ccPelBuffer(ccPelRoot<P>* pelroot);`

  Creates a window bound to the root image *pelroot*. Height and width are set to encompass the entire root image. Offset is set to (0,0) and the transform is set to identity.

  Parameters

pelroot	The root image the window is bound to.

  Notes

  If *pelroot* is *NULL*, the effect is the same as **ccPelBuffer()**; this window becomes unbound.

Public Member Functions

put

```
void put(const ccIPair&, P pelVal) const;
void put(c_Int32 x, c_Int32 y, P pelVal) const;
```

- `void put(const ccIPair& point, P pelVal) const;`

  Sets the value of the pixel at location *point* in image coordinates to *pelval*.

  Parameters

point	The location of the pixel in image coordinates.

pelVal The new value of the selected pixel.

Throws

ccPel::BadCoord
Argument does not correspond to a pixel within this window.

ccPel::UnboundWindow
This window is not bound to any root image.

- `void put(c_Int32 x, c_Int32 y, P pelVal) const;`

Sets the value of the pixel at location *(x, y)* in image coordinates to *pelVal*.

Parameters

x The x-coordinate of the pixel in image coordinates.

y The y-coordinate of the pixel in image coordinates.

pelVal The new value of the selected pixel.

Throws

ccPel::BadCoord
Argument does not correspond to a pixel within this window.

ccPel::UnboundWindow
This window is not bound to any root image.

pointToRow `ccPelTraits<P>::pointer pointToRow(c_Int32 y) const;`

Returns a pointer to the first pixel in row *y*, in image coordinates, of the window.

Parameters

y The row in image coordinates.

Throws

ccPel::BadCoord
Y does not correspond to a row within this window. The x value in the thrown exception object is set to 0.

ccPel::UnboundWindow
This window is not bound to any root image.

pointToPel

```
ccPelTraits<P>::pointer pointToPel(c_Int32 x, c_Int32 y)
  const;

ccPelTraits<P>::pointer pointToPel(const ccIPair& point)
  const;
```

- `ccPelTraits<P>::pointer pointToPel(c_Int32 x, c_Int32 y) const;`

  Returns a pointer to the pixel at location *(x, y)* in image coordinates.

  Parameters

  x — The x-coordinate of the pixel in image coordinates.

  y — The y-coordinate of the pixel in image coordinates.

  Throws

  ccPel::BadCoord — Argument does not correspond to a pixel within this window.

  ccPel::UnboundWindow — This window is not bound to any root image.

- `ccPelTraits<P>::pointer pointToPel(const ccIPair& point) const;`

  Returns a pointer to the pixel located at location *point* in image coordinates.

  Parameters

  point — The location of the pixel in image coordinates.

  Throws

  ccPel::BadCoord — Argument does not correspond to a pixel within this window.

  ccPel::UnboundWindow — This window is not bound to any root image.

copyFromBuffer

```
void copyFromBuffer(const P* buf) const;
```

Copies the pixels from *buf*, an array of pixels, to this window.

Buf is addressed as follows:

```
this->put(x, y, buf[x-offset().x() + (y-offset().y()) *
                    this->width()])
```

`buf[0]` corresponds to `this->put(offset())`

Notes

The parameter is **P*** rather than **ccPelTraits<P>::pointer** to allow simple pointer or array manipulation of the buffer.

Parameters

buf The array to receive the copied pixels. Must be width() * height() pixels.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

mutating

```
void mutating() const;
```

Informs the persistence system that this image's pixel values are about to be changed (or have been changed already).

Notes

Most operations that modify pixel values automatically call this function. The only time you would need to call **mutating()** explicitly is if you modify pixel values through a non-const pointer returned by **pointToRow()** or **pointToPel()**. Note that the act of obtaining such a pointer automatically calls **mutating()**.

root

```
ccPelRoot<P>* root() const;
```

Returns a pointer to this window's root image. Make sure you do not cache the returned pointer.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

disconnectRoot

```
ccPelRoot<P>* disconnectRoot();
```

Disconnects this window from its root image, causing the window to become unbound, but does not delete root image. Returns the disconnected root image.

bind

```
bool bind(const ccPelBuffer<P>& pelbuf, bool clip=false,);
```

Makes this window have the same root image as *pelbuf*. *Pelbuf*'s root offset, offset, transform, and size are ignored. If *pelbuf* is unbound, this window becomes unbound. Returns true if the result was clipped to fit within the new image; false otherwise.

This window's root offset, offset and size are not modified unless clipping is required. If *clip* is true, the result is clipped to the extent of the new root image (possibly resulting in a null window result if the window's location is entirely outside of the new root image). This window's transform is not changed.

Parameters

pelbuf The window whose root image you want to bind this window to.

clip If true, clip to the extent of the new root image.

Throws

ccPel::BadWindow

The window's location does not fit entirely within this window's new root image, and *clip* is false.

ccPelBuffer_const

```
#include <ch_cvl/pelbuf.h>

template <class P> class ccPelBuffer_const : public cc_PelBuffer;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccPelBuffer_const** class is a template class for creating rectangular windows into root images with pixels of type *P* (**ccPelRoot<P>**).This class allows read-only access to the underlying pixels. The class **ccPelBuffer** allows read/write access to the pixels. Any number of windows may refer to the same root image.

P — Template parameter specifying the value associated with each pixel. CVL provides instantiations of **ccPelRoot** for the types **c_UInt8**, **c_UInt16**, **c_UInt32**, **ccPackedRGB16Pel**, and **ccPackedRGB32Pel**.

The class **cc3PlanePelBuffer** on page 147 implements a pel buffer that uses three-plane pixels (see **cc3PlanePel** on page 145).

Functions that take pointers of type **P** as parameters or that return pointers of type **P** use the **pointer** or **const_pointer typedef** defined by **ccPelTraits** instead of **P***. See **ccPelTraits** on page 2403.

Constructors/Destructors

ccPelBuffer_const

```
ccPelBuffer_const();

ccPelBuffer_const(const ccPelRoot<P>* pelroot);
```

- `ccPelBuffer_const();`

  Creates an unbound window, that is, one that is not associated with any root image. Height and width are set to 0 and all offsets are set to (0,0). The window's transform is set to identity.

- `ccPelBuffer_const(const ccPelRoot<P>* pelroot);`

  Creates a window bound to the root image *pelroot*. Height and width are set to encompass the entire root image. Offset is set to (0,0) and the transform is set to identity.

  Parameters

  pelroot The root image the window is bound to.

  Notes

  If *pelroot* is *NULL*, the effect is the same as **ccPelBuffer_const()**; this window becomes unbound.

Public Member Functions

get

```
P get(const ccIPair& point) const;
P get(c_Int32 x, c_Int32 y) const;
```

- `P get(const ccIPair& point) const;`

  Returns the pixel at location *point* in image coordinates.

  Parameters

  point The location of the pixel in image coordinates.

  Throws

  ccPel::BadCoord
  Argument does not correspond to a pixel within this window.

  ccPel::UnboundWindow
  This window is not bound to any root image.

- `P get(c_Int32 x, c_Int32 y) const;`

  Returns the pixel at location *(x, y)* in image coordinates.

  Parameters

  x The x-coordinate of the pixel in image coordinates.

  y The y-coordinate of the pixel in image coordinates.

  Throws

  ccPel::BadCoord
  Argument does not correspond to a pixel within this window.

  ccPel::UnboundWindow
  This window is not bound to any root image.

pointToRow

```
ccPelTraits<P>::const_pointer pointToRow(c_Int32 y) const;
```

Returns a pointer to the first pixel in row *y*, in image coordinates, of the window.

Parameters

y The row in image coordinates.

Throws

ccPel::BadCoord

Y does not correspond to a row within this window. The x value in the thrown exception object is set to 0.

ccPel::UnboundWindow

This window is not bound to any root image.

---

pointToPel

```
ccPelTraits<P>::const_pointer pointToPel(
  const ccIPair& point) const;

ccPelTraits<P>::const_pointer pointToPel(
  c_Int32 x, c_Int32 y) const;
```

- ```
  ccPelTraits<P>::const_pointer pointToPel(
    const ccIPair&) const;
  ```

  Returns a pointer to the pixel located at location *point* in image coordinates.

  Parameters

  point The location of the pixel in image coordinates.

- ```
  ccPelTraits<P>::const_pointer pointToPel(
    c_Int32 x, c_Int32 y) const;
  ```

  Returns a pointer to the pixel at location *(x, y)* in image coordinates.

  Parameters

  x The x-coordinate of the pixel in image coordinates.

  y The y-coordinate of the pixel in image coordinates.

  Throws

  ccPel::BadCoord

  Argument does not correspond to a pixel within this window.

  ccPel::UnboundWindow

  This window is not bound to any root image.

pointToOffset `ccPelTraits<P>::const_pointer pointToOffset() const;`

Returns a pointer to the pixel in the upper left-hand corner of this window. Equivalent to calling **pointToPel(offset())** but more efficient.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

copyToBuffer `void copyToBuffer(P* buf) const;`

Copies the pixels from this window to *buf*, an array of pixels.

Buf is addressed as follows:

```
buf[x - offset().x() + (y - offset().y())* this->width()] =
  this->get(x, y)
```

`buf[0]` corresponds to `this->get(offset())`

Notes

The parameter is **P*** rather than **ccPelTraits<P>::pointer** to allow simple pointer or array manipulation of the buffer.

Parameters

buf The array to receive the copied pixels. Must be width() * height() pixels.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

contains `bool contains(ccPelTrait<P>::pointer ppixel) const;`

Returns true if the pixel pointed to by *ppixel* is a pixel in this window. If the window is unbound, this function returns false.

Parameters

ppixel A pointer to a pixel.

root `const ccPelRoot<P>* root() const;`

Returns a pointer to this window's root image. Do not cache the returned pointer.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

bind

```
bool bind(const ccPelBuffer_const<P>& pelbuf,
  bool clip=false);
```

Makes this window have the same root image as *pelbuf*. *Pelbuf*'s root offset, offset, transform, and size are ignored. If *pelbuf* is unbound, this window becomes unbound. Returns true if the result was clipped to fit within the new image; false otherwise.

This window's root offset, offset and size are not modified unless clipping is required. If *clip* is true, the result is clipped to the extent of the new root image (possibly resulting in a null window result if the window's location is entirely outside of the new root image). This window's transform is not changed.

Parameters

pelbuf The window whose root image you want to bind this window to.

clip If true, clip to the extent of the new root image.

Throws

ccPel::BadWindow

The window's location does not fit entirely within this window's new root image, and *clip* is false.

ccPelBuffer_const

ccPelFunc

```
#include <ch_cvl/pelfunc.h>

class ccPelFunc;
```

A name space class for global pixel-processing (**cfPel***) functions.

Enumerations

sub_mode

```
enum sub_mode;
```

This enumeration defines the ways that **cfPelSub()** can handle overflow conditions when subtracting pixel values.

Value	Meaning	8-bit Example
eSubDefault	Result wraps around.	0 - 6 ⇒ 250(-6)
eSubAbsDefault	Result is truncated and converted to signed value, then absolute value is returned.	255 - 2 ⇒ 3
eSubAbs	Yields absolute value of result.	0 - 6 ⇒ 6
eSubZero	Negative results are clamped at zero.	0 - 6 ⇒ 0
eSubShft	Result divided by 2.	0 - 6 ⇒ 253(-3)
eSubNabs	Yields negative of absolute value of result.	6 - 0 ⇒ 250(-6)
eSubLimit	Result is clamped to minimum or maximum signed value. Results greater than 127 are clamped to 127; results less than -128 are clamped to -128.	0 - 255 ⇒ -128

add_mode `enum add_mode;`

This enumeration defines the ways that **cfPelAdd()** can handle overflow conditions when adding pixel values.

Value	Meaning	8 -bit Example
eAddDefault	Result wraps around.	250 + 6 ⇒ 0
eAddAbs	Yields absolute value of result.	0 + 6 ⇒ 6
eAddClamp	Clamp to maximum pixel value.	200 + 100 ⇒ 255
eAddShft	Result divided by 2.	200 + 100 ⇒ 150
eAddLimit	Result is clamped to minimum or maximum signed value. Results greater than 127 are clamped to 127; results less than -128 are clamped to -128.	200 + 100 ⇒ 127

mult_mode `enum mult_mode;`

This enumeration defines the ways that **cfPelMult()** handles the result of multiplying pixel values.

Value	Meaning	8-bit Example
eMultDefault	In case of overflow, result wraps around.	250 * 3 ⇒ 238 (-18)
eMultShft	In all cases, the result is shifted right by 8 bits.	128 * 11 ⇒ 5

mult_add_mode `enum mult_add_mode;`

This enumeration defines the ways that **cfPelMultAdd()** handles the result of multiplying and adding pixel values.

Value	Meaning	8-bit Example
eMultAddDefault	In case of overflow, the result wraps around.	250 * 3 + 3 ⇒ 241 (-15)
eMultAddShft	In all cases, the values are multiplied, the result is shifted right by 8 bits, then the third value is added.	128 * 11 + 3 ⇒ 8

ccPelFunc

ccPelRect

```
#include <ch_cvl/pelrect.h>

class ccPelRect : public ccRectangle<c_Int32>;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes rectangular regions of pel buffers. See **ccRectangle** on page 2699 for functions to access the components of a **ccPelRect**.

Constructors/Destructors

ccPelRect

```
ccPelRect();
ccPelRect(c_Int32 w, c_Int32 h);
ccPelRect(c_Int32 x, c_Int32 y, c_Int32 w, c_Int32 h);
ccPelRect(const ccIPair& v1, const ccIPair& v2);
ccPelRect(const ccRectangle<c_Int32>& r);
```

- `ccPelRect();`

  Makes this rectangle have origin (0,0), with width and height both equal to zero.

- `ccPelRect(c_Int32 w, c_Int32 h);`

  Makes this rectangle have width *w* and height *h*. If *w* and *h* are both non-negative, the origin is (0,0).

  Parameters

w	The width of the rectangle.
h	The height of the rectangle.

Notes

If *w* is negative, the rectangle's origin x-component is *w* (negative), and its width is *-w* (positive). Similarly, if *h* is negative, the rectangle's origin y-component is *h* (negative), and its height is *-h* (positive).

- `ccPelRect(c_Int32 x, c_Int32 y, c_Int32 w, c_Int32 h);`

Makes this rectangle have the indicated origin, width, and height.

Parameters

x	The origin's x-component.
y	The origin's y-component.
w	The width of the rectangle.
h	The height of the rectangle.

Notes

If *w* is negative, the rectangle's origin x-component is *x+w*, and its width is *-w* (positive). Similarly, if *h* is negative, the rectangle's origin y-component is *y+h,* and its height is *-h* (positive).

- `ccPelRect(const ccIPair& v1, const ccIPair& v2);`

Makes this rectangle the minimum enclosing rectangle of the indicated two points.

Parameters

v1	The origin (upper left component)
v2	The lower right component

- `ccPelRect(const ccRectangle<c_Int32>& r)`

Copies the upper left and bottom right values of rectangle *r* into this rectangle.

Parameters

r	The rectangle whose values will be copied into this rectangle.

Operators

operator= `ccPelRect& operator=(const ccRectangle<c_Int32>& r);`

Copies the upper left and lower right values of rectangle *r* into this rectangle.

Parameters

r The rectangle whose values will be copied into this rectangle.

operator+ `ccPelRect operator+ (const ccPelRect& se) const;`

Returns a rectangle dilated by the amount specified in the structuring element *se*. The dilation of a region by a structuring element is the union of all points covered by the structuring element over all positions of its origin within the region.

The identity structuring element has an upper left value of (0,0) and a size of (1,1). Dilating a rectangle by this structuring element leaves the rectangle unchanged.

Parameters

se The structuring element.

Notes

Dilation by a null structuring element is always null.

operator+= `ccPelRect& operator+= (const ccPelRect& se);`

Dilate this rectangle by the structuring element *se*.

Parameters

se The structuring element.

operator- `ccPelRect operator- (const ccPelRect& se) const;`

Returns a rectangle eroded by the amount specified in the structuring element *se*. The erosion of a region by a structuring element *se* is the set of all positions of the origin of *se* over all the positions of *se* that fit within the region.

The identity structuring element has an upper left value of (0,0) and a size of (1,1). Eroding a rectangle by this structuring element leaves the rectangle unchanged.

Parameters

se The structuring element.

Notes

Erosion by a null structuring element is undefined.

operator-= `ccPelRect& operator-= (const ccPelRect& se);`

Erode this rectangle by the structuring element *se*.

Parameters

se The structuring element.

Notes

Erosion by a null structuring element is undefined.

ccPelRoot

```
#include <ch_cvl/pelbuf.h>
#include <ch_cvl/coloroot.h>

template <class P> class ccPelRoot : public cc_PelRoot;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

The **ccPelRoot** class is a template class for creating root images with pixels of type **P**.

P — Template parameter specifying the value associated with each pixel. CVL provides instantiations of **ccPelRoot** for the types **c_UInt8**, **c_UInt16**, **c_UInt32**, **ccPackedRGB16Pel**, **ccPackedRGB32Pel**, and **cc3PlanePel**.

Functions that take pointers of type **P** as parameters or that return pointers of type **P** use the **pointer** or **const_pointer typedef** defined by **ccPelTraits** instead of **P***. See **ccPelTraits** on page 2403.

The file *coloroot.h* contains a specialization for **cc3PlanePel** that treats the image buffer as an RGB-like image in which the three channels are expected to be in three different planes.

Constructors/Destructors

ccPelRoot

```
ccPelRoot();

ccPelRoot(c_Int32 width, c_Int32 height,
  c_Int32 alignModulus=32);

ccPelRoot(c_Int32 width, c_Int32 height,
  ccPelTraits<P>::pointer pelArray,
  c_Int32 alignModulus=1);

ccPelRoot(c_Int32 width, c_Int32 height, c_Int32 pitch,
  ccPelTraits<P>::pointer pelArray,
  c_int32 alignModulus = 1);

virtual ~ccPelRoot();
```

- `ccPelRoot();`

  Initializes this root image to a null state with no associated pixels. The only things that you can do with a null root image are load it from a persistent archive and destroy it.

- `ccPelRoot(c_Int32 width, c_Int32 height, c_Int32 alignModulus=32);`

  Allocates contiguous storage for a root image of *width* x *height* pixels of type *P*. The pixels are default-constructed. This storage will be freed by the destructor. The byte address of the first row's first pixel is zero modulo *alignModulus*. The allocated row width (in pixels) is increased if necessary to make the row size (in bytes) a multiple of the specified *alignModulus*. Up to (*alignModulus*-1) pad pixels may be allocated for each row. Pad pixels, if any, occur to the right (greater x) of the pixels that belong to the root image. Pixel processing routines are permitted to overwrite pad pixels.

  Parameters

width	The width, in pixels, of the root image. (*width* > 0)
height	The height, in rows, of the root image. (*height* > 0)
alignModulus	A value that determines how memory used for the image is aligned. A value of 1 aligns on byte boundaries, 2 aligns on word boundaries, 4 aligns on long word boundaries, 16 aligns on 16-byte boundaries, 32 aligns on 32-byte boundaries, and so on. (*alignModulus* >= 1)

- ```
  ccPelRoot(c_Int32 width, c_Int32 height,
    ccPelTraits<P>::pointer pelArray,
    c_Int32 alignModulus=1);
  ```

  Uses *pelArray* as contiguous storage to store pixels of type *P*. *Width* and *height* refer to the logical size of the root image. The *pelArray* must have been allocated with alignment and padding as described above. You are responsible for freeing *pelArray* if necessary.

  Parameters

width	The width, in pixels, of the root image. (*width* > 0)
height	The height, in rows, of the root image. (*height* > 0)
pelArray	An array to hold width x height pixels of type *P*. (*pelArray != NULL*)
alignModulus	A value that determines how memory used for the image is aligned. A value of 1 aligns on byte boundaries, 2 aligns on word boundaries, 4 aligns on long word boundaries, 16 aligns on 16-byte boundaries, 32 aligns on 32-byte boundaries, and so on. (*alignModulus* >= 1)

- ```
  ccPelRoot(c_Int32 width, c_Int32 height, c_Int32 pitch,
    ccPelTraits<P>::pointer pelArray,
    c_int32 alignModulus = 1);
  ```

  Uses *pelArray* as contiguous storage to store pixels of type *P*. *Width* and *height* refer to the logical size of the root image. The *pelArray* must have been allocated with *pitch* pixels per row aligned by *alignModulus*. You are responsible for freeing *pelArray* if necessary.

  Parameters

width	The width, in pixels, of the root image. (*width* > 0)
height	The height, in rows, of the root image. (*height* > 0)
pitch	The number of pixels per row.
pelArray	An array to hold width x height pixels of type *P*. (*pelArray != NULL*) If *P* is **cc3PlanePel**, the three pointers in *pelArray* are assumed to be pointing to the first byte of their respective planes.
alignModulus	A value that determines how memory used for the image is aligned. A value of 1 aligns on byte boundaries, 2 aligns on word boundaries, 4 aligns on long word boundaries, 16 aligns on 16-byte boundaries, 32 aligns on 32-byte boundaries, and so on. (*alignModulus* >= 1)

- `virtual ~ccPelRoot();`

  Frees the pixel storage, if it was automatically allocated.

Public Member Functions

pels

```
ccPelTraits<P>::pointer pels() const;
```

Returns a pointer to the first pixel in the first row. See also, **ccPelBuffer::pointToRow()**.

Static Functions

padPelsNeeded

```
static c_Int32 padPelsNeeded(c_Int32 width,
  c_Int32 alignModulus);
```

Returns the number of pad pixels needed to make the row size *width* (in bytes) a multiple of *alignModulus*.

Parameters

width	The width in pixel of a root image. (*width* > 0)
alignModulus	A value that determines how memory used for the image is aligned. A value of 1 aligns on byte boundaries, 2 aligns on word boundaries, 4 aligns on long word boundaries, 16 aligns on 16-byte boundaries, 32 aligns on 32-byte boundaries, and so on. (*alignModulus* >= 1)

ccPelRootPool

```
#include <ch_cvl/pelpool.h>

class ccPelRootPool : public ccRepBase;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

The **ccPelRootPool** class is an abstract class used to work with root image pools (pel root pools). See **ccPelRoot** on page 2391 to learn more about root images.

Constructors/Destructors

This is an abstract class. To obtain a root image pool, see the global function **cfDefaultPelRootPool()()** on page 3619.

Public Member Functions

size

```
virtual size_t size() const = 0;

virtual void size(size_t bytes) = 0;
```

- `virtual size_t size() const = 0;`

  Returns the number of bytes occupied by the root image pool. This value is equal to the last value passed to **size(size_t bytes)** or to the default value (see **cfDefaultPelRootPoolSize()()** on page 3621) if it has never been called.

- `virtual void size(size_t bytes) = 0;`

  Changes the size of the of the root image pool.

  Parameters

  bytes The size of the root image pool in bytes.

  Throws

  ccPelRootPool::NotSupported
  This root image pool does not support resizing.

ccPelRootPool::Outstanding
At least one pel root is currently allocated from this pool.

flush

```
void flush();
```

Frees all of the cached **ccPelRoot** objects. When a root image is returned to its pool, the memory it occupies is cached rather than released so that later requests for a same-sized root image can be performed quickly. **flush()** releases all of the cached root images, effectively returning the pool to its initial size. **size()** invokes **flush()** before resizing the pool.

videoPelRoots

```
virtual bool videoPelRoots() const;
```

Returns true if this root image pool contains root images in video memory.

Typedefs

ccPelRootPoolPtrh

```
typedef ccPtrHandle<ccPelRootPool> ccPelRootPoolPtrh;
```

A pointer handle to a **ccPelRootPool** object.

ccPelRootPoolProp

```
#include <ch_cvl/prop.h>

class ccPelRootPoolProp;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

The pel root pool property lets you specify the source (or pool) of memory for root images (**ccPelRoot** objects).

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccPelRootPoolProp

```
ccPelRootPoolProp();

explicit ccPelRootPoolProp(
  const ccPelRootPoolPtrh& pelRootPool);
```

- `ccPelRootPoolProp();`

  Creates a new pel root pool property not associated with any FIFO. The default pel root pool is used.

- ```
  explicit ccPelRootPoolProp(
    const ccPelRootPoolPtrh& pelRootPool);
  ```

  Creates a new pel root pool property not associated with any FIFO. The specified pel root pool is used.
  ```

Parameters

pelRootPool The pool from which the acquisition FIFO gets its **ccPelRoot** objects

Public Member Functions

pelRootPool

```
const ccPelRootPoolPtrh& pelRootPool() const;
void pelRootPool(const ccPelRootPoolPtrh& pelRootPool);
```

- `const ccPelRootPoolPtrh& pelRootPool();`

  Returns the current pool from which the acquisition FIFO gets its **ccPelRoot** objects.

- `void pelRootPool(const ccPelRootPoolPtrh& pelRootPool);`

  Selects the pool from which the acquisition FIFO gets its **ccPelRoot** objects. The default pool is **cfDefaultPelRootPool()**.

  Parameters

  pelRootPool The pool from which the acquisition FIFO gets its **ccPelRoot** objects.

ccPelSpan

```
#include <ch_cvl/pelspan.h>

class ccPelSpan;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a linear region in an integer coordinate system. It is analogous to a **ccPelRect** with a height (or width) of 1. It is used to specify the destination span for the 1D image unwarp tool.

Constructors/Destructors

ccPelSpan

```
ccPelSpan();

ccPelSpan(c_Int32 origin, c_Int32 size);
```

- `ccPelSpan();`

  Constructs a zero-length **ccPelSpan**.

- `ccPelSpan(c_Int32 origin, c_Int32 size);`

  Constructs a **ccPelSpan** with the specified origin and size

  Parameters

  origin The origin.

  size The size.

  Throws

  ccRectErrors::NegativeSize
  size is less than 0.

Public Member Functions

origin

```
c_Int32 origin() const;
```

Returns the origin of this span.

size

```
c_Int32 size() const;
```

Returns the size of this span.

end

```
c_Int32 end() const;
```

Returns the end of this span.

Notes

The pel span does not "contain" the end of this span; it only contains "end()-1" (if end()-1 >= origin()).

isNull

```
bool isNull(void) const;
```

Returns true if this span is empty.

contains

```
bool contains(const ccPelSpan &that) const;
bool contains(c_Int32 val) const;
```

- `bool contains(const ccPelSpan &that) const;`

Returns true if the supplied **ccPelSpan** is contained within this **ccPelSpan**.

Parameters

that The **ccPelSpan** to evaluate.

Notes

A **ccPelSpan** always contains itself.

- `bool contains(c_Int32 val) const;`

Returns true if the indicated position is contained within this **ccPelSpan**.

Parameters

val The position to evaluate.

Operators

operator+

```
ccPelSpan operator+ (const ccPelSpan& se) const;
```

Returns a **ccPelSpan** that is the dilation of this **ccPelSpan** by the supplied structuring element (a **ccPelSpan**).

The new object's origin will be equal to the sum of the origins of the supplied objects, and the new object's size will be equal to the sum of the sizes of the supplied objects minus 1.

Parameters

se	The **ccPelSpan** to use as a structuring element. *se* must be non-null.

Notes

If either of the **ccPelSpan**s is empty, then the sum will be the default constructed **ccPelSpan**.

operator+=

```
ccPelSpan& operator+= (const ccPelSpan& se);
```

Dilates this **ccPelSpan** by the supplied structuring element (a **ccPelSpan**).

The dilated object's origin will be equal to the sum of the origins of the supplied objects, and the dilated object's size will be equal to the sum of the sizes of the supplied objects minus 1.

Parameters

se	The **ccPelSpan** to use as a structuring element. *se* must be non-null.

Notes

If either of the **ccPelSpan**s is empty, then the sum will be the default constructed **ccPelSpan**.

operator-

```
ccPelSpan operator- (const ccPelSpan& se) const;
```

Returns a **ccPelSpan** that is the erosion of this **ccPelSpan** by the supplied structuring element (a **ccPelSpan**).

The new object's origin will be equal to the difference of the origins of the supplied objects, and the new object's size will be equal to the differences of the sizes of the supplied objects plus 1.

Parameters

se	The **ccPelSpan** to use as a structuring element. *se* must be non-null.

Notes

If this **ccPelSpan** is smaller than the supplied **ccPelSpan**, the result is a default-constructed **ccPelSpan**.

operator-= `ccPelSpan& operator-= (const ccPelSpan& se);`

Erodes this **ccPelSpan** by the supplied structuring element (a **ccPelSpan**).

The eroded object's origin will be equal to the difference of the origins of the supplied objects, and the erorded object's size will be equal to the difference of the sizes of the supplied objects plus 1.

Parameters

se	The **ccPelSpan** to use as a structuring element. *se* must be non-null.

Notes

If this **ccPelSpan** is smaller than the supplied **ccPelSpan**, the result is a default-constructed **ccPelSpan**.

Erode this pelspan by the specified structuring element.

operator== `bool operator== (const ccPelSpan& that) const;`

Returns true if the supplied object is equal to this one, false otherwise

Parameters

that	The object to compare.

ccPelTraits

```
#include <ch_cvl/peltrait.h>

template <class P> class ccPelTraits;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	No

The **ccPelTraits** class is a template class used to describe pointer and reference types for all of the different pixel types used by **ccPelBuffer** and its related classes. For example, **ccPelRoot<P>::pels()** returns a pointer to the first pixel in the first row of the pel root. Instead of returning **P***, which may not be appropriate for types P, the function returns a value of type **ccPelTraits<P>::pointer** which is guaranteed to be a pointer for any support pixel type.

CVL provides instantiations of **ccPelTraits** for the following classes:

- **c_UInt8**
- **c_UInt16**
- **c_UInt32**
- **ccPackedRGB16Pel**
- **ccPackedRGB32Pel**
- **cc3PlanePel**

Static Functions

isColor

```
static bool isColor();
```

Returns true if the pixel class represents color information

isPacked

```
static bool isPacked();
```

Returns true if the storage for a single pixel is guaranteed to be contiguous.

Typedefs

pointer

```
typedef P* pointer;
```

A pointer to a non-const object

reference

```
typedef P& reference;
```

A reference to a non-const object

const_pointer

```
typedef const P *const_pointer;
```

A pointer to a const object

const_reference

```
typedef const P & const_reference;
```

A reference to a const object

ccPerimPos

```
#include <ch_cvl/shapbase.h>

class ccPerimPos;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class implements perimeter positions. Perimeter positions represent arbitrary and unique points along a **ccShape** in terms of a distance (arc length) along the shape from the *first point*. Both the first point on the shape and the manner in which the shape is traversed as a function of arc length, are consistent with the way the **sample()** operation is defined to work on the shape.

Perimeter ranges, which comprise a perimeter position and a signed distance, can be used to delineate arbirtrary portions of a shape. See **ccPerimRange** on page 2406.

See the *Shapes* chapter of the *CVL User's Guide* for more information on perimeter positions.

Note You never need to construct **ccPerimPos** objects explicitly. They are created internally when you call certain **ccShapeInfo** and **ccShape** methods.

Operators

operator==

```
bool operator==(const ccPerimPos& other) const;
```

Returns true if and only if this perimeter position is exactly equal to the given perimeter position.

Parameters

other The other perimeter position.

operator!=

```
bool operator==(const ccPerimPos& other) const;
```

Returns true if and only if this perimeter position is not exactly equal to the given perimeter position.

Parameters

other The other perimeter position.

operator< `bool operator<(const ccPerimPos& other) const;`

Returns true if and only if the given perimeter position is further along the perimeter of a shape than this perimeter position, in the direction of shape traversal.

Parameters

other The other perimeter position.

operator> `bool operator>(const ccPerimPos& other) const;`

Returns true if and only if this perimeter position is further along the perimeter of a shape than the given perimeter position, in the direction of shape traversal.

Parameters

other The other perimeter position.

Typedefs

ccPerimRange `typedef cmStd pair<ccPerimPos, double> ccPerimRange;`

This type definition specifies a perimeter range. Perimeter ranges, which comprise a perimeter position and a signed distance, can be used to delineate arbirtrary portions of a shape.

Notes

A perimeter range begins at the given perimeter position, and extends the given distance along the perimeter of the shape. The range may extend in either direction, depeding on the sign of the distance.

ccPersistent

```
#include <ch_cvl/persist.h>

class ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Not applicable

The **ccPersistent** class is a base class from which you derive complex-persistent objects. To implement a complex-persistent class, you must follow these guidelines:

1. Your class must be derived using the **virtual** keyword from **ccPersistent**. (In a complex derivation hierarchy there must be only a single instance of the **ccPersistent** base class.)
2. Your class declaration must begin with the **cmPersistentDcl** macro. (Use the **cmPersistentDclTemplate** macro for template classes and the **cmPersistentDclAbstract** macro for abstract base classes.)
3. Your class implementation must begin with the **cmPersistentDef** macro. (Use the **cmPersistentDefTemplate** macro for template classes and the **cmPersistentDefAbstract** macro for abstract base classes.)
4. Your class must override the **ccPersistent::serialize_()** function. This function takes a **ccArchive** as its first argument. Your implementation of this function must persist the state of the object to the supplied **ccArchive**.
5. Whenever your class's state changes, the class must call the **ccPersistent::mutating()** function.

For more information on writing complex-persistent objects, refer to the files *ch_cvl\persist.h* and *sample\persist.cpp*.

Constructors/Destructors

ccPersistent

```
ccPersistent();

virtual ~ccPersistent();

ccPersistent(const ccPersistent&);
```

Do not instantiate this class directly.

Operators

operator=

```
ccPersistent& operator= (const ccPersistent& rhs);
```

Assignment operator.

Parameters

rhs The **ccPersistent** to assign from.

operator||

```
ccArchive& operator|| (ccArchive& arc, ccPersistent& obj);

template <class T>
 ccArchive& operator|| (ccArchive& arc, T *& ptr);

template <class T>
 ccArchive& operator|| (ccArchive& arc, const T * ptr);
```

- `ccArchive& operator|| (ccArchive& arc, ccPersistent& obj);`

  This overload of **operator||** performs non-polymorphic serialization of the supplied object. This supports serialization of complex-persistent objects embedded as members of other persistent objects but does not support serialization through a pointer.

  When you load a complex-persistent object using this overload of **operator||**, the object into which you are loading the persisted object must already exist (since you pass a reference to it to this function). The persistence system calls the object's **reInit()** function before it serializes the data. Since you can override **reInit()** in your complex-persistent class, you can use your implementation of **reInit()** to place the object in a known state before the data is serialized.

  This behavior is different than the **operator||** overloads that support pointer-based serialization (listed below). When you load a complex-persistent operator using the pointer-based **operator||** overload, an instance of the object is created using the **new** operator, then it is serialized.

  You can ensure that the results of pointer-based and reference-based loading of complex-persistent objects are identical by having your object's default constructor call your override of **reInit()**.

  Parameters

  arc The archive into which the object is serialized.

  obj The **ccPersistent**-derived object being serialized.

Throws

ccArchive::BadType

The **ccArchive** loaded an invalid dynamic object type (loading only). The archive's position is unspecified after this error.

ccArchive::UnknownType

The **ccArchive** did not recognize the loaded dynamic object type (loading only). The archive's position is unspecified after this error.

ccArchive::UnknownVersion

A persistent object's serialization code did not recognize a data version number (loading only).

ccArchive::Eof

No more input data (loading only). The archive is left positioned at the end of the input stream.

ccArchive::BadBuffer

The **ccArchive** is broken and is unusable (loading or storing).

Notes

When storing, this function may only be called in the context of sequential stores onto the end of this archive. When loading, the object being loaded must have been stored through a reference serialization (rather than by use of pointer serialization or **ccArchive::storeObject()**).

Your override of the **serialize_()** function in your **ccPersistent**-derived class can assume that whenever the object is being loading from an archive, it is loaded into a default-constructed object or an object upon which **reInit()** has been called.

- ```
  template <class T>
  ccArchive& operator|| (ccArchive& arc, const T * ptr);
  ```

  This overload of **operator||** performs polymorphic serialization of the supplied object through a pointer. This overload is intended for use in *storing* a complex-persistent object through a pointer; you pass the pointer to the **operator||** function.

  Complex-persistent objects stored through a pointer are annotated with type information allowing subsequent loading through a pointer to dynamically allocate an object of the appropriate dynamic type. The archive tracks what objects have been serialized through pointers in order to properly handle structural sharing of dynamically allocated complex-persistent objects.

  When storing, the pointed-to object is stored immediately (if it has not been stored already).

  When storing, this function may only be called in the context of sequential stores onto the end of this archive.
  ```

Parameters

arc The archive into which the object is serialized

ptr A pointer to the **ccPersistent**-derived object being serialized.

Throws

ccArchive::NotFound
A pointed-to object could not be located in this archive (loading only). The archive is positioned after the pointer.

ccArchive::UnknownType
The **ccArchive** did not recognize the loaded dynamic object type (loading only). The archive is positioned after the pointer.

ccArchive::UnknownVersion
The loaded persistent object's serialization code did not recognize a data version number (loading only). The archive is positioned after the pointer.

ccArchive::Eof
No more input data (loading only). The archive is left positioned at the end of the input stream.

ccArchive::BadBuffer
The **ccArchive** is broken and is unusable (loading or storing).

If one of the above errors occurs when loading the pointed-to object, the loaded pointer is set to null.

Notes

Serializing a pointer to a complex-persistent member sub-object of another persistent object is not well supported; when storing a pointer to a complex-persistent member sub-object of another object, only the referenced sub-object is stored. The entire parent object is not stored. When the pointer is loaded, only the pointed-to sub-object is loaded.

When storing through a pointer, the pointed-to complex persistent object need not be dynamically allocated. In such cases, be aware that the stored object will be dynamically allocated when it is loaded. Therefore, storing a pointer to an object that is not dynamically allocated should be done with care.

- ```
  template <class T>
  ccArchive& operator|| (ccArchive& arc, T *& ptr);
  ```

  This overload of **operatorII** performs polymorphic serialization through a reference to a pointer to the object. This overload is primarily intended for use in *loading* a complex-persistent object through a pointer; you pass a reference to the pointer to the **operatorII** function. (The overload will store the object if the archive's mode is storing.)

  When you load a complex-persistent object using this overload of **operatorII**, an instance of the object is dynamically allocated using the **new** operator, then the stored data is serialized into the new object, and the pointer that you supplied is set to point to the new object.

  This behavior is different than the **operatorII** overload that supports reference-based (non-polymorphic) serialization. When you load a complex-persistent operator using the non-polymorphic **operatorII** overload, the object instance already exists. The persistence system calls that object's overload of **reInit()** to allow you to initialize the object to a known state before serialization.

  You can ensure that the results of pointer-based and reference-based loading of complex-persistent objects are identical by having your object's default constructor call your override of **reInit()**.

  When loading, the object being loaded must have been stored through pointer serialization (rather than by use of reference serialization or **ccArchive::storeObject()**).

  Parameters

  arc — The archive into which the object is serialized

  ptr — A pointer to the **ccPersistent**-derived object being serialized.

  Throws

  ccArchive::NotFound
  A pointed-to object could not be located in this archive (loading only). The archive is positioned after the pointer.

  ccArchive::UnknownType
  The **ccArchive** did not recognize the loaded dynamic object type (loading only). The archive is positioned after the pointer.

  ccArchive::UnknownVersion
  The loaded persistent object's serialization code did not recognize a data version number (loading only). The archive is positioned after the pointer.

  ccArchive::Eof
  No more input data (loading only). The archive is left positioned at the end of the input stream.
  ```

ccArchive::BadBuffer
The **ccArchive** is broken and is unusable (loading or storing).

If one of the above errors occurs when loading the pointed-to object, the loaded pointer is set to null.

Notes

When storing a pointer, the pointed-to complex persistent object need not be dynamically allocated. In such cases, be aware that the stored object will be dynamically allocated when it is loaded. Therefore, storing a pointer to an object that is not dynamically allocated should be done with care.

operator<<

```
friend ccArchive& operator<< (ccArchive& ar,
  ccPersistent const& obj);

template <class T>
friend ccArchive& operator<< (ccArchive& ar,
  const T * ptr);
```

These overloads of the **operator<<** function are identical to the corresponding **operator||** functions except that the **ccArchive** must be storing.

operator>>

```
friend ccArchive& operator>> (ccArchive& ar,
  ccPersistent& obj);

friend ccArchive& operator>> (ccArchive& ar,
  T *& ptr);
```

These overloads of the **operator>>** function are identical to the corresponding **operator||** functions except that the **ccArchive** must be loading.

Protected Member Functions

reInit

```
virtual void reInit();
```

This function is called prior to loading into an existing complex-persistent object.

This function can be overridden to prepare an object to be loaded. Typically, this function will be overridden to put the object back into its default-constructed state, so that the object's **serialize_()** implementation may assume that it is always loading into a object with a known state.

The base class implementation has no effect.

Notes

In general, a derivation's **reInit()** override should also call its direct base class's **reInit()** function.

loadSimple

```
virtual bool loadSimple(ccArchive& ar);
```

If this class used to be simple persistent, then this function should be overridden to do the following:

1) Call **reInit()**.

2) Perform the old simple persistent loading.

3) Return true.

The base class implementation simply returns false.

Parameters

ar The archive to load.

Notes

The persistence mechanism will call this function only if it has already determined that the object was stored in its old simple persistent form. Thus the override does not need to do any special checking, and it may throw as appropriate.

haveVisited

```
static bool haveVisited(void*, size_t);
```

Called by **dispatchSerialize()**. Queries whether the indicated base sub-component of the currently serializing **ccPersistent** object has been visited.

Returns 0 if this component has already been visited while traversing the currently serializing **ccPersistent** object.

Returns 1 if this component has not yet been visited. Requires the indicated sub-component to be within the currently serializing **ccPersistent** object.

Notes

This function is for Cognex internal use only.

setVisited

```
static void setVisited(void*, size_t);
```

Called by **dispatchSerialize()**. Records that the indicated base sub-component of the currently serializing **ccPersistent** object has been visited.

Requires the indicated sub-component to be within the currently serializing **ccPersistent** object.

Notes

This function is for Cognex internal use only.

mutating

```
void mutating();
```

Notifies any archives that may care about this instance that this instance is either being destroyed or is about to have its state altered.

Notes

This function should not be called by **serialize_()**.

Private Member Functions

serialize_

```
void serialize_(ccArchive& ar, c_UInt8 dataVersion);
```

Each complex-persistent class that you derive from **ccPersistent** must define an implementation for this function (the derivation macros provide the declaration). Your implementation should perform the following steps:

- For each direct base class derived (directly or indirectly) from **ccPersistent**, call that class's **dispatchSerialize()** function.
- For each direct base class *not* derived from **ccPersistent**, serialize that base class (using that class's override of **operator||**).
- Serialize each member of your class using **operator||**.

Parameters

ar	The archive to or from which the data is serialized.
dataVersion	The data version. You specify the data version as an argument to the derivation macro. Whenever your persistent class changes, you should increment this value. Keep in mind that your **serialize_()** function must be prepared to handle both current and older data versions.

Notes

The base class implementation does nothing.

Your implementation of this function must *not* call **mutating()**.

When storing, *dataVersion* is always the current data version number for this class; when loading, *dataVersion* might be different from the current data version number. In such cases, your **serialize_** implementation must either correctly process the different (presumably older) data format, or it should throw *ccArchive::UnknownVersion*.

This function must not call any protected functions of **ccPersistent** other than **dispatchSerialize()**.

ccPersistent

ccPMAlignDefs

```
#include <ch_cvl/pmalign.h>

class ccPMAlignDefs : public cc_PMDefs;
```

A name space that holds enumerations and constants used with PatMax.

Enumerations

TrainMethod `enum TrainMethod`

This enumeration defines the training methods supported by PatMax.

Value	Meaning
kImageTrainMethod = 1	Train using an image model.
kSyntheticTrainMethod = 2	Train using a synthetic model. **Note**: This method had been deprecated.
kShapeTrainMethod = 3	Train using a geometric shape.
kFeatureTrainMethod = 4	Train using edge features.

DrawMode `enum DrawMode`

This enumeration defines the types of result graphics you can draw for PatMax results.

Value	Meaning
eDrawOrigin = 0x1	Draws a cross at the location of the result. The cross is aligned to the angle at which the pattern is found.
eDrawLabel = 0x2	Draws a text label at the result location.
eDrawBoundingBox = 0x4	Draws a box around the region of the image that corresponds to the trained pattern.
eDrawWireFrame = 0x8	Draws a wireframe representation of the found pattern.
eDrawMatchInfo = 0x10	Draws the matched feature boundary points.
eDrawMatchInfoPattern = *eDrawMatchInfo*	Draws the matched feature boundary points.

Value	Meaning
eDrawMatchInfoImage = 0x20	Draws the run-time image feature match info.
eDrawStandard = 0xf	Draws a labeled, pattern-aligned cross at the pattern location along with a box that corresponds to the trained pattern and a wireframe representation of the trained pattern.

ccPMAlignPattern

```
#include <ch_cvl/pmalign.h>

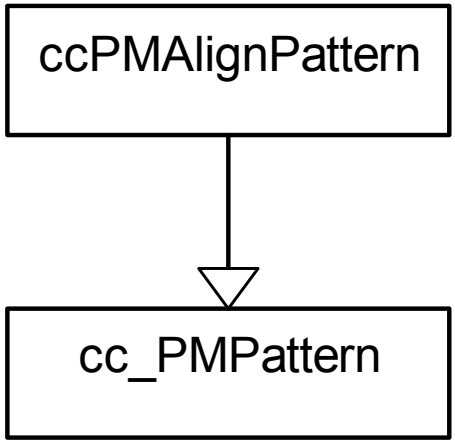
class ccPMAlignPattern : public cc_PMPattern;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single trained PatMax pattern. This class includes functions to set the training parameters and a function to locate the pattern in an image.

The following figure shows the **ccPMAlignPattern** class inheritance hierarchy.



Note See the **cc_PMPattern** class for information on additional functions common to all PatMax pattern classes.

Constructors/Destructors

ccPMAlignPattern

```
ccPMAlignPattern();
~ccPMAlignPattern();
ccPMAlignPattern(const ccPMAlignPattern& rhs);
```

- `ccPMAlignPattern();`

  Construct an untrained patmax alignment pattern with default values

- `~ccPMAlignPattern();`

  Destructor.

- `ccPMAlignPattern(const ccPMAlignPattern& rhs);`

  Copy constructor. Constructs an object equal to *rhs*.

  If **rhs.trainMethod()** = **ccPMAlignDefs::ImageTrainMethod()** and **rhs.saveImage()** is false, the resulting pattern is not trained regardless of whether *rhs* is currently trained.

Operators

operator= `ccPMAlignPattern& operator=(const ccPMAlignPattern& rhs);`

Assignment operator. Set this object equal to *rhs*.

If **rhs.trainMethod()** is equal to *ccPMAlignDefs::kImageTrainMethod* and **rhs.saveImage()** is false, the resulting pattern is not trained regardless of whether *rhs* is currently trained.

Public Member Functions

ignorePolarity

```
virtual bool ignorePolarity() const;
virtual void ignorePolarity(bool t);
```

- `virtual bool ignorePolarity() const;`

  Retrieves the *ignorePolarity* flag.

Notes

Overrides the virtual method inherited from the **cc_PMPattern** base class.

- `virtual void ignorePolarity(bool t);`

Sets the *ignorePolarity* flag.

Parameters

t If true, PatMax will not use polarity as one of the search criteria.

Throws

ccPMAlignDefs::NotImplemented

The parameter *t* is false, and any portion of the shape on which this pattern was trained has an effective ignore polarity setting of true.

Notes

This function overrides the virtual function inherited from the **cc_PMPattern** base class. Its behavior is the same as that of the base class method, with the exception of the throw.

See also **cc_PMPattern::ignorePolarity()**.

---

saveImage

```
bool saveImage() const;

void saveImage(bool save);
```

Used with image training. If you are training PatMax with an image, this function controls whether a copy of the training image pixels are stored in this pattern during training (default = true).

This function is ignored unless the tool is trained with an image.

- `bool saveImage() const;`

Returns true if this **ccPMAlignPattern** is currently configured to save an internal copy of the pattern training image, otherwise false. If this function returns false when **train()** is called, image pixels are not saved.

Returned values are valid only if you are working with an image model.

- `void saveImage(bool save);`

  Control whether a copy of the training image pixels are stored in this pattern when it is trained with an image. If this function returns false when **train()** is called, image pixels are not saved.

  If you change **saveImage()** from true to false on a trained pattern, saved image pixels are discarded. **saveImage()** cannot be changed from false to true on a pattern that has been trained with an image. Any attempts to do this are ignored.

  Calls to this function are ignored if the object is trained with a **ccShape**.

  Parameters

  save Set to true to save a copy of the pattern training image (default = true).

  Throws

  ccPMAlignDefs::BadParams
  If **saveImage(true)** is called on a trained pattern for which **isImageTrainMethod()** is true and **saveImage()** is currently false.

  Notes

  Setting **saveImage()** to false has limitations with respect to copying, assigning, and archiving. See the comments of those functions for details.

---

train

```
void train(
 const ccPelBuffer_const<c_UInt8>& image,
 c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
 ccDiagObject* diagobj = 0,
 c_UInt32 diagFlags=0);

void train(
 const ccPelBuffer_const<c_UInt8>& image,
 const ccPelBuffer_const<c_UInt8>& mask,
 c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
 ccDiagObject* diagobj=0,
 c_UInt32 diagFlags=0);

void train(
 const ccShape& trainShape,
 const cc2XformBase& clientFromImage,
 const ccPelRect& trainRegion = ccPelRect(0,0),
```

```
c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
ccDiagObject* diagobj = 0,
c_UInt32 diagFlags = 0);
```

- ```
  void train(
    const ccPelBuffer_const<c_UInt8>& image,
    c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0);
  ```

  Trains this **ccPMAlignPattern** using the supplied pattern training image and the supplied algorithms. You can perform a search using only an algorithm for which the **ccPMAlignPattern** has been trained.

  Parameters

  image — The pattern training image. Unless you have previously called **saveImage()** with the *save* argument set to false, this function saves an internal copy of *image*.

  algorithms — The algorithms for which to train this **ccPMAlignPattern**. *algorithms* must be formed by ORing together one or more of the following values. Note that neither *kPatflex* nor *kPatpersp* can be used with another algorithm.

  ccPMAlignDefs::kPatquick
  ccPMAlignDefs::kPatmax
  ccPMAlignDefs::kPatflex
  ccPMAlignDefs::kPatpersp

  diagobj — An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.

  diagFlags — The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccPMAlignDefs::BadParams
  If *algorithms* is invalid, or if *algorithms* has selected both the *cc_PMDefs::kPatflex* flag and another flag. When PatFlex is
  ```

trained the pattern cannot be used to train another algorithm.
OR
If *algorithms* has both the *cc_PMDefs::kPatpersp* flag and another flag. When PatPersp is trained, the pattern cannot be used to train another algorithm.

ccPMAlignDefs::CanNotTrain
If *image* is not bound,
or if *image* does not contain any features,
or if *image* contains a NULL window,

Call the exception's **message()** function to determine the specific problem.

ccEdgeletDefs::CoordOutOfRange
If the pattern size is greater than 32769 pixels in width or height.

Notes

If this **ccPMAlignPattern** is already trained, it is automatically untrained and retrained.

This function supports the use of a **ccTimeout**-based timeout.

- ```
  void train(
    const ccPelBuffer_const<c_UInt8>& image,
    const ccPelBuffer_const<c_UInt8>& mask,
    c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0);
  ```

Trains this **ccPMAlignPattern** using the supplied pattern training image, the supplied algorithms, and the supplied mask. You can perform a search using only an algorithm for which the **ccPMAlignPattern** has been trained.

The mask image must have the same dimensions and offset as the pattern training image. The pixels in the mask image are interpreted as follows:

- All pixels in *image* that correspond to pixels in *mask* with values greater than or equal to 192 are considered 'care' pixels. All feature boundary points detected within 'care' pixels are included in the trained pattern.
- All pixels in *image* that correspond to pixels in *mask* with values from 0 through 63 are considered 'don't care but score' pixels. Feature boundary points detected within 'don't care but score' pixels are not included in the trained pattern.

  When the trained pattern is located in a run-time image, features within the 'don't care but score' part of the trained pattern are treated as clutter features.
  ```

- All pixels in *image* that correspond to pixels in *mask* with values from 64 through 127 are considered 'don't care and don't score' pixels. Feature boundary points detected within 'don't care and don't score' pixels are not included in the trained pattern.

  When the trained pattern is located in a run-time image, features within the 'don't care and don't score' part of the trained pattern are ignored and not treated as clutter features.

- Mask pixel values from 128 through 191 are reserved for future use by Cognex.

Parameters

image — The pattern training image. Unless you have previously called **saveImage()** with the *save* argument set to false, this function saves an internal copy of *image*.

mask — The mask image.

algorithms — The algorithms for which to train this **ccPMAlignPattern**. *algorithms* must be formed by ORing together one or more of the following values. Note that neither *kPatflex* nor *kPatpersp* can be used with another algorithm.

ccPMAlignDefs::kPatquick
ccPMAlignDefs::kPatmax
ccPMAlignDefs::kPatflex
ccPMAlignDefs::kPatpersp

diagobj — An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.

diagFlags — The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMAlignDefs::BadParams
If *algorithms* is invalid, or if *algorithms* has selected both the *cc_PMDefs::kPatflex* flag and another flag. When PatFlex is trained the pattern cannot be used to train another algorithm.
OR

If *algorithms* has both the *cc_PMDefs::kPatpersp* flag and another flag. When PatPersp is trained, the pattern cannot be used to train another algorithm.

ccPMAlignDefs::CanNotTrain
If *image* or *mask* is not bound,
or if *image* does not contain any features,
or if *image* contains a NULL window,
or it *mask* contains reserved grey levels,
or if *mask* does not have the same offset and dimensions as *image*.

Call the exception's **message()** function to determine the specific problem.

ccEdgeletDefs::CoordOutOfRange
If the pattern size is greater than 32769 pixels in width or height.

Notes

If this **ccPMAlignPattern** is already trained, it is automatically untrained and retrained.

This function supports the use of a **ccTimeout**-based timeout.

- ```
  void train(
    const ccShape& trainShape,
    const cc2XformBase& clientFromImage,
    const ccPelRect& trainRegion = ccPelRect(0,0),
    c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
    ccDiagObject* diagobj = 0,
    c_UInt32 diagFlags = 0);
  ```

Trains this pattern from the given shape, defined in training client coordinates. The relationship between the client and image space of the pattern at train time is specified by the *clientFromImage* transform. The specified *trainRegion* rectangle indicates the effective extent of the trained pattern in training image coordinates. If a null rectangle is specified (the default), the tool will automatically compute the effective training image region as the bounding box of the trained pattern (which may actually be larger than the specified train shape) in training image coordinates. If the trained pattern in training image coordinates does not lie completely within the specified image region, it will be clipped. Otherwise, as long as the trained pattern lies completely within the specified region, the exact rectangle is only important if *scoreUsingClutter* is true when the tool is run, since it represents the region over which the clutter will be computed. All of the algorithms specified by *algorithms* are trained. *algorithms* is the bit-wise OR of one or more of *ccPMAlignDefs::Algorithms*. Only algorithms that are trained are available at run time.
  ```

Parameters

trainShape The shape to train.

clientFromImage
The client space from image space transform.

trainRegion The rectangular region in training image coordinates where the training takes place.

algorithms The algorithms for which to train this **ccPMAlignPattern**. *algorithms* must be formed by ORing together one or more of the following values. Note that neither *kPatflex* nor *kPatpersp* can be used with another algorithm.

ccPMAlignDefs::kPatquick
ccPMAlignDefs::kPatmax
ccPMAlignDefs::kPatflex
ccPMAlignDefs::kPatpersp

diagobj An optional **ccDiagObject**. If you supply a value, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMAlignDefs::CanNotTrain
The shape contains too few features when mapped into image space by the supplied client transform and within the effective training region.

ccPMAlignDefs::BadParams
If *algorithms* is invalid, or if *algorithms* has selected both the *cc_PMDefs::kPatflex* flag and another flag. When PatFlex is trained the pattern cannot be used to train another algorithm.
OR

If *algorithms* has both the *cc_PMDefs::kPatpersp* flag and another flag. When PatPersp is trained, the pattern cannot be used to train another algorithm.

ccPMAlignDefs::BadWeights
None of the effective weights within the supplied shape and within the effective train region are > 0.

ccPMAlignDefs::ModelTooLarge
shape.isFinite() is false, and **trainRegion.isNull()** is true.

ccPMAlignDefs::NotImplemented
The effective ignore polarity flag for any portion of the supplied shape is true, and the **ignorePolarity()** setting for this pattern is false.

ccEdgeletDefs::CoordOutOfRange
If the pattern size is greater than 32769 pixels in width or height.

Notes

If this pattern is already trained, it is automatically untrained and then retrained using the given parameters.

When polarity is not ignored, dark to light transitions are expected when moving from *negative* to *positive* sides of shape boundaries, as defined by the effective polarity of the shape, after the pattern has been aligned with the run-time image. In other words, negative means dark and positive means light.

All portions of the shape (including those with 0 weight) are used to automatically compute the training region, if unspecified. At run time, however, portions are treated differently depending on their weight. Note that the weights within a shape are relative. That is, scaling each of the weights by the same factor does not affect runtime behavior, as they are internally normalized.

Weights are treated as follows:

- **Positive effective weight:** Results favor those poses for which the portion matches the image features with high accuracy, although the particular value of the weight has no effect on accuracy. The portion contributes positively to the PatQuick result score in an amount proportional to the normalized weight,

to the perimeter of the portion, and to the degree of match at the result pose. Image features that match the portion are not considered clutter. Such features instead contribute to the coverage score, but the particular value of the weight is not used when computing coverage.

- **Negative effective weight:** Results favor those poses for which the portion does not match the image at all. The portion has no effect on the accuracy of the returned pose. The portion contributes negatively to a PatQuick result score in an amount proportional to the normalized weight, to the perimeter of that portion, and to the degree of match at the result pose. Image features that match the portion do not contribute to the coverage score. Such features are considered clutter.
- **Zero effective weight:** The portion has no effect on the result pose or its accuracy, and contributes nothing to the PatQuick result score, regardless of the degree of match. Image features that match the portion do not contribute to the coverage score. Such features are instead considered clutter.

As the particular value of a weight does not affect the coverage or clutter scores, a PatMax algorithm result score is only affected by the sign of a weight. Only the sign determines whether matched features contribute to clutter or coverage. Therefore, if you want to experiment with weighting schemes to optimize performance with respect to the acceptance threshold, the PatMax algorithm (if used) should be temporarily disabled. The returned score will then be the PatQuick result score, which is affected by the particular values of the weights, and not just their signs. When the PatMax algorithm is again enabled, the PatQuick score will continue to be internally computed and compared to the acceptance threshold, but will generally no longer be returned as the score of the result. Instead, the PatMax algorithm score will be returned.

See the **ccShapeModel** and **ccShapeModelProps** reference pages for information about model boundaries and polarities. See the *PatMax* and *Shape Models* chapters of the *CVL Vision Tools Guide* for more information about the weight property.

Notes

PatMax ignores the *magnitude* properties of the supplied shape.

trainAdvanced

```
void trainAdvanced(
   const ccPelBuffer_const<c_UInt8>& image,
   c_UInt32 algorithms = ccPMAlignDefs::kPatmax,
   ccDiagObject* diagobj=0, c_UInt32 diagFlags=0);

void trainAdvanced(
   const ccPelBuffer_const<c_UInt8>& image,
   const ccPelBuffer_const<c_UInt8>& mask,
   c_UInt32 algorithms = ccPMAlignDefs::kPatmax,
   ccDiagObject* diagobj=0, c_UInt32 diagFlags=0);
```

-
```
void trainAdvanced(
   const ccPelBuffer_const<c_UInt8>& image,
   c_UInt32 algorithms = ccPMAlignDefs::kPatmax,
   ccDiagObject* diagobj=0, c_UInt32 diagFlags=0);
```

The usage, parameters, and throws of this function are identical to those of the first overload of **train()**, as described on on page 2423, with the following amendment for the selected algorithm:

Throws

ccPMAlignDefs::NotImplemented
If the selected algorithm is not *cc_PMDefs::kPatmax*

This function performs additional processing to optimize the selection of the coarse granularity limit. The optimized training can produce better results when used with pattern training images that contain repeating features.

-
```
void trainAdvanced(
   const ccPelBuffer_const<c_UInt8>& image,
   const ccPelBuffer_const<c_UInt8>& mask,
   c_UInt32 algorithms = ccPMAlignDefs::kPatmax,
   ccDiagObject* diagobj=0, c_UInt32 diagFlags=0);
```

The usage, parameters, and throws of this function are identical to those of the second overload of **train()**, as described on on page 2424, with the following amendment for the selected algorithm:

Throws

ccPMAlignDefs::NotImplemented
If the selected algorithm is not *cc_PMDefs::kPatmax*

This function performs additional processing to optimize the selection of the coarse granularity limit. The optimized training can produce better results when used with pattern training images that contain repeating features.

untrain

```
void untrain();
```

Untrains this **ccPMAlignPattern**. Discards any saved pattern training image. If this **ccPMAlignPattern** is not trained, this function has no effect.

run

```
void run(const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPMAlignRunParams& params,
  class ccPMAlignResultSet& resultSet,
  ccDiagObject* diagobj = 0, c_UInt32 diagFlags = 0) const;

void run(const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPMAlignRunParams& params,
  const cc2XformBasePtrh_const& startPose,
  class ccPMAlignResultSet& resultSet,
  ccDiagObject* diagobj=0, c_UInt32 diagFlags=0) const;

void run(const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccPMAlignRunParams& params,
  class ccPMAlignResultSet& resultSet,
  ccDiagObject* diagobj = 0, c_UInt32 diagFlags = 0) const;

void run(const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccPMAlignRunParams& params,
  const cc2XformBasePtrh_const& startPose,
  class ccPMAlignResultSet& resultSet,
  ccDiagObject* diagobj=0, c_UInt32 diagFlags=0) const;
```

-
```
void run(const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPMAlignRunParams& params,
  class ccPMAlignResultSet& resultSet,
  ccDiagObject* diagobj = 0,
  c_UInt32 diagFlags=0) const;
```

Locate this **ccPMAlignPattern** in the supplied image using the parameters in the supplied **ccPMAlignRunParams**. The supplied **ccPMAlignResultSet** is cleared and then filled with the results of the search, in decreasing order of score.

Parameters

inputImage	The image in which to locate this pattern.
params	The search parameters to use.
resultSet	A reference to a **ccPMAlignResultSet** into which to place the results of this search.
diagobj	An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMAlignDefs::NotTrained
This model is not trained.

ccPMAlignDefs::BadImage
If *inputImage* is unbound.

ccPMAlignDefs::BadParams
params specifies an algorithm for which this **ccPMAlignPattern** is not trained.

ccTimeout::Expired
The tool did not find a pattern instance within the timeout specified.

ccPMAlignDefs::InternalError
This error can occur if the tool locates a pattern in *inputImage* that is displaced by more than 10,000 client units or is less than one tenth the size or greater than ten times the size of the trained pattern. If this is the case, this exception's **message()** function returns the string **Scale and/or Offset out of range**.

ccPMAlignDefs::NotImplemented
If *algorithm* is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp* and *sensitivityMode* is *kSensitivityModeHigh*.

ccPMAlignDefs::NotImplemented
If *algorithm* is *cc_PMDefs::kPatflex* and *outsideRegionThreshold* is > 0.

Notes

This function terminates if it runs longer than the timeout specified in *params*. If this occurs, the tool results are invalid.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& inputImage,
    const ccPMAlignRunParams& params,
    const cc2XformBasePtrh_const& startPose,
    class ccPMAlignResultSet& resultSet,
    ccDiagObject* diagobj=0, c_UInt32 diagFlags=0) const;
  ```

  Locate this **ccPMAlignPattern** in the supplied image using the parameters in the supplied **ccPMAlignRunParams**.

  The search is started at the pose specified by *startPose*. The supplied pose is refined in each degree of freedom that is enabled and has a nonzero range in the supplied **ccPMAlignRunParams** object. This override is intended for use in cases where you already have an accurate coarse location for the pattern instance.

  The supplied **ccPMAlignResultSet** is cleared and then filled with the results of the search, in decreasing order of score.

  Parameters

  inputImage — The image in which to locate this pattern.

  params — The search parameters to use.

  startPose — A reference to a **cc2Xform** indicating where to start the search.

  resultSet — A reference to a **ccPMAlignResultSet** into which to place the results of this search.

  diagobj — An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

  diagFlags — The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccPMAlignDefs::NotTrained
  This model is not trained.

  ccPMAlignDefs::BadImage
  If *inputImage* is unbound,.
  ```

ccPMAlignDefs::BadParams
: *params* specifies an algorithm for which this **ccPMAlignPattern** is not trained or *params* specifies an algorithm of *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatmax* and *startPose* is a nonlinear transform.

ccTimeout::Expired
: The tool did not find a pattern instance within the timeout specified.

ccPMAlignDefs::NotImplemented
: If *algorithm* is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp* and *sensitivityMode* is *kSensitivityModeHigh*.

Notes

This function terminates if it runs longer than the timeout specified in *params*. If this occurs, the tool results are invalid.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& inputImage,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccPMAlignRunParams& params,
    class ccPMAlignResultSet& resultSet,
    ccDiagObject* diagobj = 0,
    c_UInt32 diagFlags = 0) const;
  ```

  Locate this **ccPMAlignPattern** in the supplied image using the parameters in the supplied **ccPMAlignRunParams**. The image is masked using the supplied mask image. Pixels in the mask image with values less than 128 are *don't care* pixels. Pixels in the mask image with values greater than or equal to 128 are *care* pixels. All feature boundary points in the supplied image that lie within pixels that correspond to *don't care* pixels in the mask image are discarded.

  The supplied **ccPMAlignResultSet** is cleared and then filled with the results of the search, in decreasing order of score.

  Parameters

  inputImage — The image in which to locate this pattern.

  mask — The mask image. *mask* must have the same dimensions and image offset as *inputImage.*

  params — The search parameters to use.

  resultSet — A reference to a **ccPMAlignResultSet** into which to place the results of this search.

  diagobj — An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.
  ```

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMAlignDefs::NotTrained
This model is not trained.

ccTimeout::Expired
The tool did not find a pattern instance within the timeout specified.

ccPMAlignDefs::BadParams
params specifies an algorithm for which this **ccPMAlignPattern** is not trained.

ccPMAlignDefs::BadImage
inputImage or *mask* is unbound or *inputImage* and *mask* do not have the same dimensions and image offset.

ccPMAlignDefs::InternalError
This error can occur if the tool locates a pattern in *inputImage* that is displaced by more than 10,000 client units or is less than one tenth the size or greater than ten times the size of the trained pattern. If this is the case, this exception's **message()** function returns the string **Scale and/or Offset out of range**.

ccPMAlignDefs::NotImplemented
If *algorithm* is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp* and *sensitivityMode* is *kSensitivityModeHigh*.

ccPMAlignDefs::NotImplemented
If *algorithm* is *cc_PMDefs::kPatflex* and *outsideRegionThreshold* is > 0.

Notes

This function terminates if it runs longer than the timeout specified in *params*. If this occurs, the tool results are invalid.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& inputImage,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccPMAlignRunParams& params,
    const cc2XformBasePtrh_const& startPose,
    class ccPMAlignResultSet& resultSet,
    ccDiagObject* diagobj=0, c_UInt32 diagFlags=0) const;
  ```

  Locate this **ccPMAlignPattern** in the supplied image using the parameters in the supplied **ccPMAlignRunParams**. The image is masked using the supplied mask image. Pixels in the mask image with values less than 128 are *don't care* pixels. Pixels in the mask image with values greater than or equal to 128 are *care* pixels. All feature boundary points in the supplied image that lie within pixels that correspond to *don't care* pixels in the mask image are discarded.

  The search is started at the pose specified by *startPose*. The supplied pose is refined in each degree of freedom that is enabled and has a nonzero range in the supplied **ccPMAlignRunParams** object. This override is intended for use in cases where you already have an accurate coarse location for the pattern instance.

  The supplied **ccPMAlignResultSet** is cleared and then filled with the results of the search, in decreasing order of score.

  Parameters

inputImage	The image in which to locate this pattern.
mask	The mask image. *mask* must have the same dimensions and image offset as *inputImage.*
params	The search parameters to use.
startPose	A reference to a **cc2Xform** indicating where to start the search.
resultSet	A reference to a **ccPMAlignResultSet** into which to place the results of this search.
diagobj	An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.
diagFlags	The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*
  ```

Throws

ccPMAlignDefs::NotTrained
This model is not trained.

ccPMAlignDefs::BadImage
inputImage or *mask* is unbound or *inputImage* and *mask* do not have the same dimensions and image offset.

ccPMAlignDefs::BadParams
params specifies an algorithm for which this **ccPMAlignPattern** is not trained or *params* specifies an algorithm of *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatmax* and *startPose* is a nonlinear transform.

ccTimeout::Expired
The tool did not find a pattern instance within the timeout specified.

ccPMAlignDefs::NotImplemented
If *algorithm* is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp* and *sensitivityMode* is *kSensitivityModeHigh*.

Notes

This function terminates if it runs longer than the timeout specified in *params*. If this occurs, the tool results are invalid.

trainImage

```
ccPelBuffer_const<c_UInt8> trainImage() const;
```

Returns the pattern training image used to train this **ccPMAlignPattern**.

Throws

ccPMAlignDefs::NotTrained
This model is not trained.

ccPMAlignDefs::NoImage
The pattern training image was not saved when this **ccPMAlignPattern** was trained, or was discarded because you called **saveImage()** with the *save* argument set to false after training.

ccPMAlignDefs::WrongTrainMethod
trainMethod() is not equal to *ccPMAlignDefs::kImageTrainMethod*.

maskImage

```
ccPelBuffer_const<c_UInt8> maskImage() const;
```

Returns the mask image used to train this **ccPMAlignPattern**.

Throws

ccPMAlignDefs::NotTrained
This model is not trained.

ccPMAlignDefs::NoImage
saveImage() was not called with a value of true for this **ccPMAlignPattern**.

ccPMAlignDefs::WrongTrainMethod
trainMethod() is not equal to *ccPMAlignDefs::kImageTrainMethod*.

Notes

Only the training time mask is saved. The run-time mask is not available.

trainShape

```
const ccShape& trainShape() const;
```

Returns the geometric shape used to train this pattern.

Throws

ccPMAlignDefs::NotTrained
This pattern is not trained.

ccPMAlignDefs::WrongTrainMethod
trainMethod() is not equal to *ccPMAlignDefs::kShapeTrainMethod*.

trainRegion

```
ccPelRect trainRegion() const;
```

Returns image region used to train this pattern.

Throws

ccPMAlignDefs::NotTrained
This pattern is not trained.

ccPMAlignDefs::WrongTrainMethod
trainMethod() is equal to *ccPMAlignDefs::kSyntheticTrainMethod*. (*kSyntheticTrainMethod* is deprecated)

trainMethod

```
ccPMAlignDefs::TrainMethod trainMethod() const;
```

Returns the training method used to train this **ccPMAlignPattern**.

Throws

ccPMAlignDefs::NotTrained
This model is not trained.

isImageTrainMethod

```
bool isImageTrainMethod() const;
```

Returns true if **trainMethod()** is equal to *ccPMAlignDefs::kImageTrainMethod*, otherwise false.

Throws

ccPMAlignDefs::NotTrained
This pattern is not trained.

Protected Member Functions

assignUntrainedPatternData_

```
void assignUntrainedPatternData_(ccPMAlignUntrainedData&
ptmxUtData) const;
```

Copies information in *this to **ccPMAlignUntrainedData**

Deprecated Members

train

```
void train(
  const cmStd vector<cc2Wireframe>& trainShapes,
  const cc2Xform clientFromImage,
  c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);

void train(
  const cmStd vector<cc2Wireframe>& trainShapes,
  const cmStd vector<double>& trainWeights,
  const cc2Xform clientFromImage,
  c_UInt32 algorithms = ccPMAlignDefs::kPatquick,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);
```

trainShapes

```
const cmStd vector<cc2Wireframe>& trainShapes() const;
```

trainWeights

```
const cmStd vector<double>& trainWeights() const;
```

Friends

ccPMMultiModel

```
friend class ccPMMultiModel;
```

ccPMAlignPattern

ccPMAlignResult

```
#include <ch_cvl/pmalign.h>

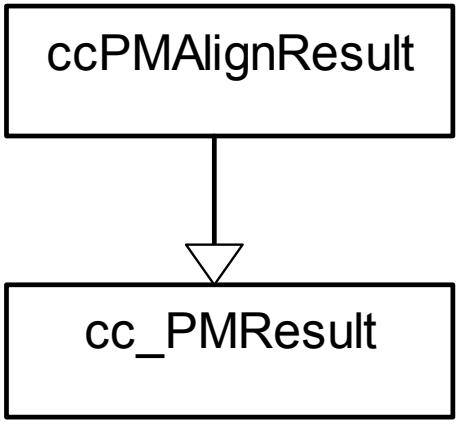
class ccPMAlignResult : public cc_PMResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single PatMax/Align search result. You should not create a **ccPMAlignResult** directly.

The following figure shows the ccPMAligResult class inheritance hierarchy.



Note See the **cc_PMResult** class for additional functions common to all PatMax result classes.

Constructors/Destructors

ccPMAlignResult

```
ccPMAlignResult();
```

Default constructor. Constructs a **ccPMAlignResult** containing no meaningful information.

Public Member Functions

draw

```
void draw(ccGraphicList& graphList,
  const ccPMAlignResultSet& set,
  c_UInt32 drawMode=ccPMAlignDefs::eDrawStandard,
  const ccCvlString& label=ccCvlString()) const;
```

Appends result graphics for this **ccPMAlignResult** to the supplied **ccGraphicList**.

Parameters

graphList — The graphics list to append the graphics to.

set — You must supply a reference to a **ccPMAlignResultSet** that contains this **ccPMAlignResult**.

drawMode — The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccPMAlignDefs::eDrawOrigin
ccPMAlignDefs::eDrawLabel
ccPMAlignDefs::eDrawBoundingBox
ccPMAlignDefs::eDrawWireFrame
ccPMAlignDefs::eDrawMatchInfo
ccPMAlignDefs::eDrawMatchInfoPattern
ccPMAlignDefs::eDrawStandard

label — If you include *ccPMAlignDefs::eDrawLabel* in *drawMode*, then the contents of *label* are used to label the center of mass.

Notes

Graphics are drawn in **ccColor::greenColor()** if the result met your acceptance threshold, **ccColor::redColor()** if it did not.

All graphics are drawn in client coordinates.

ccPMAlignResultSet

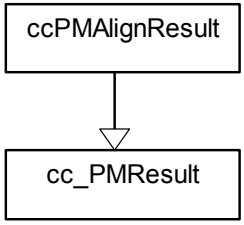
```
#include <ch_cvl/pmalign.h>

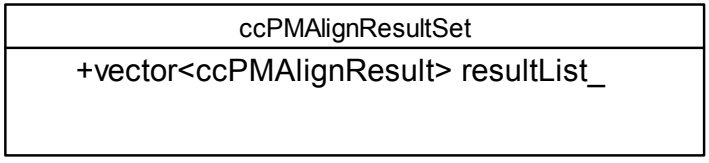
class ccPMAlignResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains all of the **ccPMAlignResult** objects returned by a PatMax alignment.





Constructors/Destructors

ccPMAlignResultSet

```
ccPMAlignResultSet();

virtual ~ccPMAlignResultSet();

ccPMAlignResultSet(const ccPMAlignResultSet&);
```

- `ccPMAlignResultSet();`

  Creates a **ccPMAlignResultSet** with no results. **numFound()** and **time()** both return 0 when called on a default-constructed **ccPMAlignResultSet**.

Public Member Functions

time

```
double time() const;
```

Returns the amount of time in seconds required to perform this alignment.

numFound

```
c_Int32 numFound() const;
```

Returns the number of **ccPMAlignResult**s in this **ccPMAlignResultSet**.

algorithm

```
ccPMAlignDefs::Algorithm algorithm() const;
```

Returns the algorithm which was used to locate the results in this **ccPMAlignResultSet**. This function returns one of the following values:

ccPMAlignDefs::kPatquick
ccPMAlignDefs::kPatmax

getResult

```
void getResult(c_Int32 index, ccPMAlignResult& result)
  const;
```

Places the contents of the requested result in the supplied **ccPMAlignResult** reference.

Parameters

index	The index of the result to retrieve.
result	A reference to a **ccPMAlignResult** into which to place the result.

Throws

ccPMAlignDefs::BadParams
index is equal or greater than **numFound()** or *index* is less than 0.

results

```
const cmStd vector<ccPMAlignResult>& results() const;
```

Returns a vector of **ccPMAlignResult**s that contains all the results from this alignment. The results in the returned vector are sorted in decreasing order of score.

infoStrings

```
cmStd vector<cccVlString> infoStrings() const;
```

Returns a vector of run-time diagnostic message strings. For additional information about the diagnostic messages, see the *CVL Vision Tools Guide*.

infoIds

```
cmStd vector<c_UInt32> infoIds() const;
```

Returns a vector of run-time diagnostic message IDs. For a list and description of the diagnostic message IDs, see the *CVL Vision Tools Guide*.

draw

```
void draw(ccGraphicList& graphList,
  c_UInt32 drawMode = ccPMAlignDefs::eDrawStandard,}
  bool drawOnlyAccepted = false) const;
```

Appends result graphics for this **ccPMAlignResultSet** to the supplied **ccGraphicList**.

Parameters

graphList The graphics list to append the graphics to.

drawMode The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccPMAlignDefs::eDrawOrigin
ccPMAlignDefs::eDrawLabel
ccPMAlignDefs::eDrawBoundingBox
ccPMAlignDefs::eDrawWireFrame
ccPMAlignDefs::eDrawMatchInfo
ccPMAlignDefs::eDrawStandard

drawOnlyAccepted
If true, only results which met your accept threshold are drawn. If false, all results are drawn.

ccPMAlignResultSet

ccPMAlignRunParams

```
#include <ch_cvl/pmalign.h>

class ccPMAlignRunParams : public cc_PMRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters used to control a Patmax alignment.

Note See the **cc_PMRunParams** class for additional functions common to all PatMax run-time parameter classes.

Constructors/Destructors

ccPMAlignRunParams

```
ccPMAlignRunParams();

~ccPMAlignRunParams();

ccPMAlignRunParams(const ccPMAlignRunParams&);
```

- `ccPMAlignRunParams();`

  Constructs a **ccPMAlignRunParams** with default parameter values.

Public Member Functions

algorithm

```
ccPMAlignDefs::Algorithm algorithm() const;

void algorithm(ccPMALignDefs::Algorithm alg);
```

- `ccPMAlignDefs::Algorithm algorithm() const;`

  Returns the algorithm that this **ccPMAlignRunParams** is configured to use. The function returns one of the following values:

  ccPMAlignDefs::kPatquick
  ccPMAlignDefs::kPatmax

- `void algorithm(ccPMALignDefs::Algorithm alg);`

  Sets the algorithm that this **ccPMAlignRunParams** uses to perform alignments.

  Parameters

  alg The algorithm to use. *alg* must be one of the following values:

  ccPMAlignDefs::kPatquick
  ccPMAlignDefs::kPatmax

saveMatchInfoModes

```
void saveMatchInfoModes (c_UInt32 modes);

c_UInt32 saveMatchInfoModes () const;
```

- `void saveMatchInfoModes (c_UInt32 modes);`

  Sets modes indicating whether to save information necessary to generate a match display, modes are set as a bitwise-OR of values from *ccPMAlignDefs::SaveMatchInfoMode*

  The following modes are available (described in **cc_PMDefs**):

  ccPMAlignDefs::kSaveMatchInfoModeNone

  ccPMAlignDefs::kSaveMatchInfoModeClassic

  ccPMAlignDefs::kSaveMatchInfoModePattern

  ccPMAlignDefs::kSaveMatchInfoModeImage

  The default is *ccPMAlignDefs::kSaveMatchInfoModeNone*

  Notes

  If the modes contain anything other than *kSaveMatchInfoModeNone*, some extra time and significant extra memory is used for each result returned. The modes are ignored if the algorithm is *kPatquick*, in which case no information is saved.

  Parameters

  modes The modes to set.

- `c_UInt32 saveMatchInfoModes () const;`

  Returns modes indicating whether to save information necessary to generate a match display, modes are returned as a bitwise-OR of values from *ccPMAlignDefs::SaveMatchInfoMode*

Deprecated Members

saveMatchInfo

```
void saveMatchInfo (bool save);
bool saveMatchInfo () const;
```

- `void saveMatchInfo (bool save);`

  Determines whether this **ccPMAlignRunParams** is configured to generate and store match information.

  Parameters

  save — If save is true, match information is generated; if save is false, no match information is generated.

  Notes

  Match information is not saved when you archive a **ccPMAlignResult**.

  Saving match information greatly increases PatMax's memory usage. Alignments where match information is generated and stored take somewhat longer than alignments where match information is not generated and stored.

  Match information is only generated and stored if the alignment is performed using the *ccPMAlignDefs::kPatmax* algorithm.

  Notes

  This function has been deprecated and is provided for backwards compatibility only. New code should use **saveMatchInfoModes()** instead.

  saveMatchInfo(*false***)** is equivalent to **saveMatchInfoModes(***kSaveMatchInfoModeNone***)**, and **saveMatchInfo(***true***)** is equivalent to **saveMatchInfoModes(***kSaveMatchInfoModeClassic***)**.

- `bool saveMatchInfo () const;`

  Returns true if this **ccPMAlignRunParams** is configured to generate and store match information. Match information is required to produce a diagnostic display showing which pattern features were matched in the run-time image. The function **ccPMAlignResult::displayMatch()** produces this diagnostic display.

  Notes

  This function has been deprecated and is provided for backwards compatibility only. New code should use **saveMatchInfoModes()** instead.

  This function returns true if **saveMatchInfoModes()** has the *kSaveMatchInfoModeClassic* and/or *kSaveMatchInfoModePattern* bits set, and false otherwise.

ccPMCompositeModelDefs

```
#include <ch_cvl/pmcmpmod.h>

class ccPMCompositeModelDefs;
```

A name space that holds an enumeration used with the Composite Model of the PatMax tool.

Enumerations

TrainInstanceOverflowHandling

```
enum TrainInstanceOverflowHandling;
```

This enumeration defines the train instance overflow handling policy. That is, if additional instances are added beyond *maxNumTrainInstances*, this property determines the behavior of **ccPMCompositeModelManager::addTrainInstance()** and **ccPMCompositeModelManager::addTrainInstances()**.

Value	Meaning
eThrow = 0	The **ccPMCompositeModelManager::addTrainInstance()** or **ccPMCompositeModelManager::addTrainInstances()** function throws *ccPMAlignDefs::CanNotTrain* if the total number of train instances would become greater than *maxNumTrainInstances*
eIgnore = 1	Train instances beyond *maxNumTrainInstances* are silently ignored.
eFIFO = 2	The least-recently-added train instance(s) are discarded in favor of the most-recently-added train instance(s) to maintain a total of no more than *maxNumTrainInstances* For *eFIFO* to operate as intended, **saveImages()** must be true.

ccPMCompositeModelDefs

ccPMCompositeModelManager

```
#include <ch_cvl/pmcmpmod.h>

class ccPMCompositeModelManager;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class is used to manage, create, train, and incrementally train Composite Models for PatMax.

Constructors/Destructors

ccPMCompositeModelManager

```
ccPMCompositeModelManager();
```

Default constructor.

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

saveImages

```
bool saveImages() const;

void saveImages(bool save);
```

- `bool saveImages() const;`

  Returns whether train images and related information are saved in the model manager.

- `void saveImages(bool save);`

  Sets whether train images and related information are saved in the model manager.

  When set to true, the original images and pose transforms from all **addTrainInstance()** or **addTrainInstances()** calls are saved in the model manager and can be retrieved using **getTrainInstances()**.

  When set to false, only the data that is required for producing Composite Models is saved, and **getTrainInstances()** will return empty vectors.

  The default is true.

  Parameters

  save If true, train images and related information are saved in the model manager.

  Notes

  Changing the value of **saveImages()** removes all train instances, but does not change parameter values.
  It is recommended that **saveImages()** only be set before training has begun.

  If you wish to be able to manually adjust grain limits in the future (by calling **retrainGrainLimits()**), **saveImages()** must be set to true.

maxNumTrainInstances

```
c_Int32 maxNumTrainInstances() const;

void maxNumTrainInstances(c_Int32 maxInstances);
```

- `c_Int32 maxNumTrainInstances() const;`

  Returns the maximum number of training instances.

- `void maxNumTrainInstances(c_Int32 maxInstances);`

  Sets the maximum number of training instances.

  Parameters

  maxInstances The maximum number of training instances.

  The default is 8.

Notes

If additional instances are added beyond this number, the resulting behavior is defined by **ccPMCompositeModelDefs::trainInstanceOverflowHandling**.

The run-time of **produceCompositeModel()** grows more than linearly in the number of train instances, so setting this value too large may result in extremely slow performance.

maxNumTrainInstances cannot be changed if the training has started.

Throws

ccPMCompositeModelDefs::BadParams
maxInstances is less than 2.

ccPMAlignDefs::NotImplemented
The setter would change the value and **isTrainStarted()** is true.

trainInstanceOverflowHandling

```
ccPMCompositeModelDefs::TrainInstanceOverflowHandling
trainInstanceOverflowHandling() const;

void trainInstanceOverflowHandling(
ccPMCompositeModelDefs::TrainInstanceOverflowHandling
policy);
```

- ```
  ccPMCompositeModelDefs::TrainInstanceOverflowHandling
  trainInstanceOverflowHandling() const;
  ```

  Returns the train instance overflow handling policy.

- ```
  void trainInstanceOverflowHandling(
  ccPMCompositeModelDefs::TrainInstanceOverflowHandling
  policy);
  ```

  Sets the train instance overflow handling policy.

  Parameters

  policy The train instance overflow handling policy.

  The default is *ccPMCompositeModelDefs::eThrow*

  Notes

  trainInstanceOverflowHandling can be changed even if training has started, and will take effect the next time instances are added.

Throws

ccPMCompositeModelDefs::BadParams

policy is not a valid enum value for **ccPMCompositeModelDefs::TrainInstanceOverflowHandling** or *ccPMCompositeModelDefs::eFIFO* (for **ccPMCompositeModelDefs::TrainInstanceOverflowHandling**) is attempted while **saveImages()** is false.

startTrain

```
void startTrain(
const ccPMAlignPattern& initialTrainedModel,
ccDiagObject* diagobj = 0,
c_UInt32 diagFlags = 0);
```

Initializes training with a trained PatMax model.

Parameters

initialTrainedModel
The initial trained model.

diagobj A diagnostic object.

diagFlags Diagnostic flags.

Notes

Some of the settings on this model are used throughout Composite Model training, such as granularities.
Composite Model will not perform well if *initialTrainedModel* is trained with the *kPatquick* algorithm only.

isTrainStarted

```
bool isTrainStarted() const;
```

Returns true if there is currently an *initialTrainedModel* from a call to **startTrain()**.

The default is false.

addTrainInstance

```
void addTrainInstance(
const ccPelBuffer_const<c_UInt8>& image,
const cc2XformLinear& modelFromClient,
ccDiagObject* diagobj = 0,
c_UInt32 diagFlags = 0);
```

Processes a single train instance to be used by the next call to **produceCompositeModel()**.

The *image* parameter is the input image containing an instance of the desired model.

The *modelFromClient* parameter specifies the pose that maps the image's client coordinates to a common model space.

Parameters

image The input image.

modelFromClient The pose that maps the image's client coordinates to a common model space.

diagobj A diagnostic object.

diagFlags Diagnostic flags.

Throws

ccPMAlignDefs::CanNotTrain
Any of the images contains no usable features,
or **isTrainStarted()** is false,
or the number of train instances would become greater than *maxNumTrainInstances* and **trainInstanceOverflowHandling()** is equal to *ccPMCompositeModelDefs::eThrow,*
or the number of train instances would become greater than *maxNumTrainInstances* and **trainInstanceOverflowHandling()** is equal to *ccPMCompositeModelDefs::eFIFO* and **saveImages()** is false.

Notes

The edge extraction parameters and granularities used are from the *ccPMAlignPattern* passed into **startTrain()**.

addTrainInstances

```
void addTrainInstances(
const cmStd vector<ccPelBuffer_const<c_UInt8> >& imageVec,
const cmStd vector<cc2XformLinear>& modelFromClientVec,
ccDiagObject* diagobj = 0,
c_UInt32 diagFlags = 0);
```

Processes train instances to be used by the next call to **produceCompositeModel()**.

The *imageVec* parameter is the set of input images containing instances of the desired model.

The *modelFromClientVec* parameter corresponds by index to *imageVec* and specifies the poses that map each image's client coordinates to a common model space.

Parameters

imageVec The set of input images.

modelFromClientVec
The poses that map each image's client coordinates to a common model space.

diagobj A diagnostic object.

diagFlags Diagnostic flags.

Throws

ccPMAlignDefs::CanNotTrain
Any of the images contains no usable features,
or **isTrainStarted()** is false,
or the number of train instances would become greater than *maxNumTrainInstances* and **trainInstanceOverflowHandling()** is equal to *ccPMCompositeModelDefs::eThrow*,
or the number of train instances would become greater than *maxNumTrainInstances* and **trainInstanceOverflowHandling()** is equal to *ccPMCompositeModelDefs::eFIFO* and **saveImages()** is false.

ccPMCompositeModelDefs::BadParams
imageVec and *modelFromClientVec* are not the same size or if either is empty.

Notes

The edge extraction parameters and granularities used are from the *ccPMAlignPattern* passed into **startTrain()**.

getTrainInstances

```
void getTrainInstances(
cmStd vector<ccPelBuffer_const<c_UInt8> >& imageVec,
cmStd vector<cc2XformLinear>& modelFromClientVec);
```

Fills the supplied vectors with all train images and transforms, including from the model used for **startTrain()**.

Parameters

imageVec The set of train images.

modelFromClientVec
The transforms.

Notes

Empty vectors are returned if **saveImages()** is false.

reset

```
void reset();
```

Removes all train instances, including the **startTrain()** data, and frees all associated memory. All parameters are restored to default-constructed state.

Notes

Consecutive calls to **reset()** have no effect.

produceCompositeModel

```
void produceCompositeModel(
const ccPMCompositeModelParams& params,
ccPMAlignPattern& resultCompositeModel,
ccDiagObject* diagobj = 0,
c_UInt32 diagFlags = 0);
```

Produces a trained *resultCompositeModel* from all accumulated train instances, guided by the Composite Model parameters.

Parameters

params Composite Model parameters.

resultCompositeModel
The result Composite Model.

diagobj A diagnostic object.

diagFlags Diagnostic flags.

Notes

The last (highest index of most recently added) train instance image is used to determine some image-related parameters for the result model.

resultCompositeModel.ignorePolarity() is set to match *params.ignorePolarity*. A model produced with *params.ignorePolarity* value of true cannot be usefully run with *params.ignorePolarity* value of false, because the original direction information is lost.

The *params* parameter provides Composite-Model-specific parameters.

The *resultCompositeModel* parameter is output as the trained model that is a composite of information from the supplied instances.

The *diagobj* and *diagFlags* are the same as the parameters used by any **ccPMAlignPattern::train()** overload.

Throws

ccPMAlignDefs::CanNotTrain
isTrainStarted() is false.

retrainGrainLimits

```
void retrainGrainLimits(
double coarseGran,
double fineGran);
```

Retrains the internal data to use the new specified grain limits.

The next call to **produceCompositeModel()** will produce a result model trained with these grain limits. This requires the original images, so it throws if they have not been saved.

Parameters

coarseGran Coarse grain limit.

fineGran Fine grain limit.

Throws

ccPMAlignDefs::CanNotTrain
isTrainStarted() is false,
or **saveImages()** is false.

coarseGrainLimit

```
double coarseGrainLimit();
```

Returns the current coarse grain limit.

fineGrainLimit

```
double fineGrainLimit();
```

Returns the current fine grain limit.

ccPMCompositeModelManager

ccPMCompositeModelParams

```
#include <ch_cvl/pmcmpmod.h>

class ccPMCompositeModelParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains parameters to guide the process of creating a Composite Model.

Constructors/Destructors

ccPMCompositeModelParams

```
ccPMCompositeModelParams();
```

Default constructor.

Notes

The default destructor, copy constructor, and assignment operator are used.

Public Member Functions

minImagesFrac

```
double minImagesFrac() const;

void minImagesFrac(double frac);
```

- `double minImagesFrac() const;`

  Returns the minimum images fraction.

- `void minImagesFrac(double frac);`

  Sets the minimum images fraction.

  Parameters

  frac The minimum images fraction.

The default is 0.75.

Notes

To be considered for the final model, an edgelet must be present in at least this fraction of the images. See *ch_cvl/edge.h* for the definition of edgelet.

Be aware of how discretization works here. For example, if we have 5 images and **minImagesFrac()** is set to 50%, then a feature will actually have to be present in 60% of the images; 2 would be 40% and therefore below the threshold.

Throws

ccPMCompositeModelDefs::BadParams
frac is less than 0 or greater than 1.

matcherDistanceTolerancePels

```
double matcherDistanceTolerancePels() const;

void matcherDistanceTolerancePels(double distTolPels);
```

- `double matcherDistanceTolerancePels() const;`

Returns the internal feature matcher's maximum capture distance, in pels.

- `void matcherDistanceTolerancePels(double distTolPels);`

Sets the internal feature matcher's maximum capture distance, in pels.

Parameters

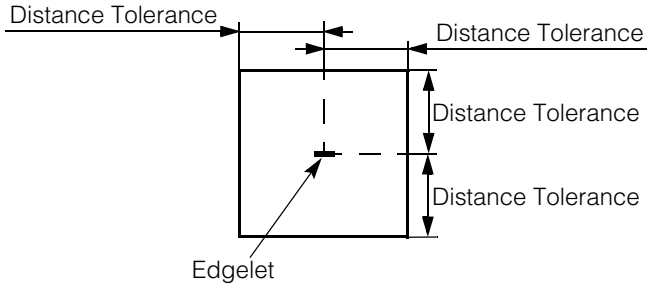
distTolPels	The internal feature matcher's maximum capture distance, in pels.

The default is 5.0.

Notes

matcherDistanceTolerancePels() forms a capture range rectangle aligned to a given edgelet.

For example, with an edgelet shown below, the capture range would look like this:



This parameter is mainly for speed, reducing the number of correspondences to consider. The internal feature matcher will always prefer closer correspondences.

It is far more convenient to specify this value in pels, so that a standard default can be meaningful for many applications. Pel units from the most recently added training instances are used.

Throws

ccPMCompositeModelDefs::BadParams
distTolPels is less than 0.

ignorePolarity

```
bool ignorePolarity() const;

void ignorePolarity(bool ignorePol);
```

- `bool ignorePolarity() const;`

  Returns the internal feature matcher's treatment of polarity.

- `void ignorePolarity(bool ignorePol);`

  Sets the internal feature matcher's treatment of polarity.

Parameters

ignorePol Internal feature matcher's treatment of polarity.

The default is false.

Notes

If *ignorePolarity* is false, then angle similarity is measured as the raw difference in edgelet angle. If *ignorePolarity* is true, then angle similarity is measured such that an opposite direction is also a perfect match. This allows a Composite Model to be built from instances whose boundaries are not of the same polarity.

If *ignorePolarity* is true, then the *ccPMAlignPattern* produced by **ccPMCompositeModelManager::produceCompositeModel()** will have its *ignorePolarity* set to true. A model produced with *ignorePolarity* value of true cannot be usefully run with *ignorePolarity* value of false, because the original direction information is lost. If *ignorePolarity* is false, then the *ccPMAlignPattern* produced by **ccPMCompositeModelManager::produceCompositeModel()** will have its *ignorePolarity* set to match that of the initial *ccPMAlignPattern* supplied to **ccPMCompositeModelManager::startTrain()**.

ccPMFlexResult

```
#include <ch_cvl/pmpbase.h>

class ccPMFlexResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single PatFlex result. You should not instantiate objects of this class in your programs. **ccPMFlexResult** objects are created for you as needed.

A description of how to run the PatFlex algorithm with PatMax is included in the introduction of the **ccPMFlexRunParams** reference page. Please see that reference.

When PatMax runs using the PatFlex algorithm it produces a PatFlex result along with its normal results. You access the PatFlex result with the getter **cc_PMResult::flexResult()**. The PatFlex result contains a deformation transform that you can apply to the run-time image to transform it into an image that matches the trained model. Cognex provides the global function **cfSampledImageWarp()** you can use for this transformation.

Constructors/Destructors

ccPMFlexResult

```
ccPMFlexResult();
```

Default constructor. Do not instantiate this class in your programs. Objects are created for you as needed.

Public Member Functions

xform

```
cc2XformBasePtrh_const xform() const;
```

Returns a transform that approximately maps the training image to the client coordinates of the run-time image.

Notes

If **computeXform()** was set to *kFitNone* in the run-time parameters, this function returns a null pointer-handle.

ccPMFlexResult

ccPMFlexRunParams

```
#include <ch_cvl/pmpbase.h>

class ccPMFlexRunParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class holds the run-time parameters used by PatFlex, a special algorithm you can run on the PatMax tool. The parameters are defaulted at construction but setters and getters are provided for you to read and write the parameters if you wish.

To run PatFlex prepare PatMax using the *kPatflex* algorithm. Add the PatFlex run-time parameters to the PatMax run-time parameters using the setter **cc_PMRunParams::flexRunParams()**. You run PatFlex by calling **ccPMAlignPattern::run()** as you would for other PatMax algorithms.

PatMax runs using the PatFlex algorithm and produces a PatFlex result along with its normal results. You access the PatFlex result with the getter **cc_PMResult::flexResult()**. See the **ccPMFlexResult** reference page for information about PatFlex results.

Constructors/Destructors

ccPMFlexRunParams

```
ccPMFlexRunParams();
```

Default constructor. The default values are:

Parameter	Default value
deformationRate	0.3
smoothness	3.0
refinement	*cc_PMDefs::kRefinementCoarse*
partialMatchMode	false
coverageThreshold	0.5

Parameter	Default value
controlPoints	(6, 6)
controlPointsExplicit	An empty vector
computeXform	*kFitDeformation*

Public Member Functions

deformationRate

```
double deformationRate() const;
void deformationRate(double rate);
```

Specifies the maximum deformation rate permitted in PatFlex alignment. The valid range is 0 through 1.0. Higher values increase the alignment time and may result in spurious matches due to the large amount of freedom the tool has to deform parts of the training image to fit the run-time image.

The default is 0.3.

- `double deformationRate() const;`

  Returns the maximum deformation rate.

- `void deformationRate(double rate);`

  Sets a new maximum deformation rate.

  Parameters

  rate The new maximum deformation rate.

  Throws

  cc_PMDefs::BadParams
  If *rate* < 0, or *rate* > 1.0.

smoothness

```
double smoothness() const;
void smoothness(double smoothness);
```

Specifies the smoothness value used to create the **cc2XformDeform** deformation transform.

The default is 3.0.

- `double smoothness() const;`

  Returns the smoothness.

- `void smoothness(double smoothness);`

  Sets the smoothness.

  Parameters

  smoothness The new smoothness.

  Throws

  cc_PMDefs::BadParams
  If *smoothness* < 0.

refinement

```
cc_PMDefs::Refinement refinement() const;
void refinement(cc_PMDefs::Refinement refinement);
```

Specifies the amount of refinement done on the deformation transform. See the following:

Refinement value	Description
kRefinementNone	No refinement. There may be small errors in the transform's mapping.
kRefinementCoarse	The computed transformation is refined so that it has no more residual error than the specified coarse granularity limit.
kRefinementMedium	The same as coarse refinement, except that certain types of high-level inaccuracy are removed.
kRefinementFine	Same as medium refinement, except that the transformation is refined so that it has no more residual error than the specified fine granularity limit.

The default is *cc_PMDefs::kRefinementCoarse*

- `cc_PMDefs::Refinement refinement() const;`

  Returns the refinement value.

- `void refinement(cc_PMDefs::Refinement refinement);`

  Sets a new refinement.

  Parameters
  refinement The new refinement.

partialMatchMode

```
bool partialMatchMode() const;
void partialMatchMode(bool partialMatchMode);
```

Specifies whether or not to use *partialMatchMode*, which accepts any candidate match for which it can match a proportion at least equal to the *coverageThreshold* with a score on those sections of the pattern greater than or equal to the *acceptThreshold*.

The default is false.

- `bool partialMatchMode() const;`

  Returns whether or not to use *partialMatchMode.*

- `void partialMatchMode(bool partialMatchMode);`

  Sets whether or not to use *partialMatchMode.*

  Parameters
  partialMatchMode
  If true, *partialMatchMode* is used.

coverageThreshold

```
double coverageThreshold() const;
void coverageThreshold(double coverageThreshold);
```

Specifies the minimum proportion of the pattern that must be matched in order to accept the match as a valid result. Has no effect if unless *partialMatchMode* is true.

The default is 0.5.

- `double coverageThreshold() const;`

  Returns the minimum proportion of the pattern that must be matched in order to accept the match as a valid result.

- `void coverageThreshold(double coverageThreshold);`

  Sets the minimum proportion of the pattern that must be matched in order to accept the match as a valid result.

  Parameters

  coverageThreshold
  The coverage threshold.

controlPoints

```
ccIPair controlPoints() const;
void controlPoints(const ccIPair &controlPoints);
```

Specifies the number of control points used in the x- and y-dimensions to define the returned deformation transform. While these points are internal to the transform, having more points allows the transform to better match sharp and more heavily deformed areas of the pattern image.

The default is (6, 6).

Notes

The returned transform is not guaranteed to have exactly the number of control points specified, nor to place them on a regular grid. The provided values are considered a guideline, but characteristics of the pattern or of the deformation may cause the tool to modify control point placement to more accurately represent the transform.

- `ccIPair controlPoints() const;`

  Returns the control points.

- `void controlPoints(const ccIPair &controlPoints);`

  Sets the control points.

  Parameters

  controlPoints The new control points.

  Throws

  cc_PMDefs::BadParams
  If either element of *controlPoints* has value less than 2.

controlPointsExplicit

```
const cmStd vector<cc2Vect> &controlPointsExplicit() const;

void controlPointsExplicit(
  const cmStd vector<cc2Vect> &pointList);
```

Specifies the exact positions of all control points used to define the returned deformation transform. An empty vector indicates that **controlPoints()** values should be used instead.

The default is an empty vector.

Notes

The control points are specified in the client coordinate system of the training image.

The control points must not all be colinear, and no two control points can have the same position.

No more than 64 control points can be specified.

- `const cmStd vector<cc2Vect> &controlPointsExplicit() const;`

  Returns the explicit control points.

- ```
  void controlPointsExplicit(
    const cmStd vector<cc2Vect> &pointList);
  ```

  Sets the explicit control points.

  Parameters

  pointList The new control points.

  Throws

  cc_PMDefs::BadParams
  If *pointList* has fewer than three elements and is not empty, or if *pointList* has more than 64 elements.
  ```

computeXform

```
cc_PMDefs::DeformationFit computeXform() const;
void computeXform(cc_PMDefs::DeformationFit computeXform);
```

Specifies the type of transform PatMax computes and returns in the PatFlex result. (See **ccPMFlexResult**). The transform type must be one of the **DeformationFit** enums as follows:

Computed transform type	Description
kFitNone	No transform is computed.
kFitDeformation	A deformation transform is computed. See **cc2XformDeformation**.
kFitPerspective	A perspective transform is computed. See **cc2XformPerspective**.

The default is *kFitDeformation*.

Notes

If you know you don't need the deformation transform from the result, you can save significant time and memory by setting **computeXform()** to *kFitNone*.

Using *kFitPerspective* can result in a more accurate transform if the run-time image is known to have only perspective deformation, or if you only want to have the perspective part of the deformation mapped.

- `cc_PMDefs::DeformationFit computeXform() const;`

  Returns the transform type.

- `void computeXform(cc_PMDefs::DeformationFit computeXform);`

  Sets the transform type. Must be one of the **DeformationFit** enums.

  Parameters

  computeXform The new transform type.

ccPMFlexRunParams

ccPMInspectAbsenceData

```
#include <ch_cvl/pminspct.h>

class ccPMInspectAbsenceData;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class stores the inspection results for the blank scene inspection mode. The boundary features stored by the class are defects in a region that is supposed to be blank (see *PatInspect* in *Vision Tools Guide*).

Constructors/Destructors

ccPMInspectAbsenceData

```
ccPMInspectAbsenceData();
```

Constructs an empty **ccPMInspectAbsenceData** object.

Public Member Functions

extra

```
const cmStd vector<ccPMInspectUnmatchedFeature>& extra()
  const;
```

Returns a reference to a vector whose elements contain defect boundary features.

Throws

ccPMInspectDefs::BadResultSetup
The function is called prior to initialization by the **ccPMInspectResult::getBlankSceneMeasurement()**.

displayFeatures

```
void displayFeatures(ccGraphicList& displayList,
  const cc2Xform& pose = cc2Xform::I,
  const ccColor& extraColor = ccColor::redColor()) const;
```

Draws the defect boundary features on *displayList*.

Parameters

displayList The graphic list used to draw the defect boundary features

pose A **cc2Xform** object used to transform the displayed boundary features

extraColor The color used to draw the defect boundary features.

Throws

ccPMInspectDefs::BadResultSetup
The function is called prior to a successful run

ccPMInspectBoundaryData

```
#include <ch_cvl/pminspct.h>

class ccPMInspectBoundaryData;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that stores boundary feature differences information. Boundary feature differences are grouped in three categories.

- Matched Features. These are boundary features that are present in both run-time and template image
- Extra Features. These are boundary features that are present in the run-time image but not in the template image.
- Missing Features. These are features that are present in the template image but not in the run-time image.

Constructors/Destructors

ccPMInspectBoundaryData

```
ccPMInspectBoundaryData();
```

Creates an empty **ccPMInspectBoundaryData** object.

Public Member Functions

match

```
const cmStd vector <ccPMInspectMatchedFeature>& match()
  const;
```

Returns the list of matched features.

Throws

ccPMInspectDefs::BadResultSetup
The function has been called prior to initialization by **ccPMInspectResult::getBoundaryDiff()**.

missing

```
const cmStd vector <ccPMInspectUnmatchedFeature>& missing()
  const;
```

Returns the list of missing features.

Throws

ccPMInspectDefs::BadResultSetup
The function has been called prior to initialization by **ccPMInspectResult::getBoundaryDiff()**.

extra

```
const cmStd vector <ccPMInspectUnmatchedFeature>& extra()
  const;
```

Returns the list of extra features.

Throws

ccPMInspectDefs::BadResultSetup
The function has been called prior to initialization by **ccPMInspectResult::getBoundaryDiff()**.

minDeformation

```
double minDeformation() const;
```

Returns the smallest deformation in client coordinates computed across all the matched boundary features.

maxDeformation

```
double maxDeformation() const;
```

Returns the biggest deformation in client coordinates computed across all the matched boundary features.

displayFeatures

```
void displayFeatures(
  ccGraphicList& displayList,
  const cc2Xform& pose = cc2Xform::I,
  bool showMatch = true,
  bool showExtra = true,
  bool showMissing = true,
  bool showConnect = true,
  ccColor matchColor = ccColor::blueColor(),
  const ccColor extraColor& = ccColor::redColor(),
  const ccColor missingColor& = ccColor::yellowColor(),
```

```
const ccColor connectColor& = ccColor::whiteColor(),
const ccColor patternColor& = ccColor::greenColor())
const;
```

Draws the difference boundary feature data (match, extra and missing) on the graphic list *displayList* using the color specified by *matchColor*, *extraColor*, *missingColor*, *connectColor*, *patternColor*.

Parameters

displayList	The graphic list to be used for drawing
pose	A **cc2Xform** object used to transform the displayed boundary features
showMatch	If true the matching boundary features are shown
showExtra	If true the extra boundary features are shown
showMissing	If true the missing boundary features are shown
showConnect	If true the connections between neighboring points on the boundary features are shown
matchColor	The color of the matching boundary features
extraColor	The color of the extra boundary features
missingColor	The color of the missing boundary features
connectColor	The color of the connections between neighboring points
patternColor	The color of the boundary features in this region

Throws

ccPMInspectDefs::BadResultSetup
The function has been called prior to initialization by **ccPMInspectResult::getBoundaryDiff()**.

Notes

This member function replaces the global function **cfPMInspectDisplayFeatures()** which will be deprecated after CVL 5.5.

Typedefs

ccPMInspectSimpleBoundaryDiffData

```
typedef ccPMInspectBoundaryData
 ccPMInspectSimpleBoundaryDiffData;
```

The class **ccPMInspectSimpleBoundaryDiffData** will be deprecated after CVL 5.5.

ccPMInspectBP

```
#include <ch_cvl/bndpnts.h>

class ccPMInspectBP;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that describes a single PatInspect feature boundary point.

Note You should not instantiate this class directly.

Constructors/Destructors

ccPMInspectBP

```
ccPMInspectBP();
```

Construct an uninitialized **ccPMInspectBP**.

Public Member Functions

pos

```
cc2Vect pos() const;
```

Returns the position of this feature boundary point in the alignment training image client coordinate system.

dir

```
ccRadian dir() const;
```

Returns the direction of this feature boundary point in the alignment training image client coordinate system. The direction is defined as the angle between the client coordinate system x-axis and the positive gradient direction of the boundary point (dark-to-light).

weight

```
double weight() const;
```

Returns the weight of this feature boundary point in the range 0.0 to 1.0. The weight is a measure of the relative strength of the boundary point in the image. The stronger the boundary point, the higher the weight.

matchQuality

```
double matchQuality() const;
```

Returns the match quality of this feature boundary point in the range -1.0 to 1.0. Match quality is a measure of the degree to which this feature boundary point matched the corresponding feature boundary point in the inspection training image and an indication of the feature boundary point's weight.

A match quality of 1.0 means that there was perfect match and that the feature boundary point had a high weight. A match quality of -1.0 means that there was a complete mismatch and that the feature boundary point had a high weight.

Match qualities close to 0 indicate that the feature boundary point had a low weight.

ccPMInspectDefs

```
#include <ch_cvl/pminspct.h>

class ccPMInspectDefs : public cc_PMDefs;
```

A name space that holds enumerations and constants used with PatInspect.

Enumerations

NormalizationMethod

```
enum NormalizationMethod
```

This enumeration defines the image normalization methods used by PatInspect (for more details see *Normalizing Run-Time Images* in *PatInspect* chapter of *Vision Tools Guide*).

Value	Meaning
eIdentity	No normalization
eHistogramEqualize	Normalize to match histograms
eMeanAndStdDev	Normalize to match mean and standard deviation
eMatchTails	Normalize to match tails

DiffMode

```
enum DiffMode
```

The difference mode specifies the pel sign handling of the difference operator.

Value	Meaning
eAbs	Absolute value of the difference pels
eClampNeg	Negative difference pels are clamped to zero. This discards negative difference pels.
eClampPos	Positive difference pels are clamped to zero and negative pels are negated. This discards positive difference pels.

Value	Meaning
eClampAdd128	Positive values are clamped to +127 and negative values are clamped to -127, after which 128 is added to all values. This preserves both positive and negative difference pels.
eDiv2Add128	All values are divided by two, then incremented by 128. This preserves both positive and negative difference pels.

InspectMode

```
enum InspectMode
```

This enumeration defines the inspection modes supported by PatInspect (to learn about each of these inspection modes see *PatInspect* chapter in *Vision Tools Guide*).

Value	Meaning
eBPDifference = 0x01	Feature difference mode
eIntensityDifference = 0x02	Intensity difference mode
eBlankScene = 0x04	Blank scene inspection mode

InspectModeModifier

```
enum InspectModeModifier
```

This enumeration defines the modifiers that can be applied to the feature difference inspection mode (for more information see *Fine Alignment for Boundary Feature Inspection* in *PatInspect* chapter of *Vision Tools Guide*).

Value	Meaning
eSoftwareZoomTrain = 0x1000	Software zoom applied only during training
eSoftwareZoomTrainRun = 0x2000	Software zoom applied during training and runtime
eFineAlign = 0x4000	Alignment to local inspection features

Valid combinations:

Inspection mode	Inspection mode modifiers
eBPDifference	*eSoftwareZoomTrain, eSoftwareZoomTrainRun, eFineAlign*
eIntensityDifference	None
eBlankScene	None

DrawMode `enum DrawMode`

Drawing modes

Value	Meaning
eDrawDefectBoundary = 0x1	Draw defect boundary features of result set when using *eBlankScene* inspection mode
eDrawMatchingBoundary = 0x2	Draw matching boundary features of result set when using *eBPDifference* inspection mode
eDrawExtraBoundary = 0x4	Draw extra boundary features of result set when using *eBPDifference* inspection mode
eDrawMissingBoundary = 0x8	Draw missing boundary features of result set when using *eBPDifference* inspection mode

ccPMInspectDefs

ccPMInspectMatchedBP

```
#include <ch_cvl/bndpnts.h>

class ccPMInspectMatchedBP : public ccPMInspectBP;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single matched PatInspect feature boundary point.

Note You should not instantiate this class directly.

Constructors/Destructors

ccPMInspectMatchedBP

```
ccPMInspectMatchedBP();
```

Construct an uninitialized **ccPMInspectMatchedBP**.

Public Member Functions

hasPatternBP

```
bool hasPatternBP() const;
```

Returns true if this feature boundary point has a corresponding feature boundary point in the inspection training image. If this function returns false, then **patternPos()**, **patternDir()**, and **patternWeight()** will not return valid information.

patternPos

```
cc2Vect patternPos() const;
```

Returns the position of this feature boundary point in the alignment training image's client coordinate system.

patternDir

```
ccRadian patternDir() const;
```

Returns the direction of this feature boundary point in the alignment training image's client coordinate system.

patternWeight `double patternWeight() const;`

Returns the weight of this feature boundary point in the inspection training image. The returned value is in the range 0.0 to 1.0.

ccPMInspectMatchedFeature

```
#include <ch_cvl/bndpnts.h>

class ccPMInspectMatchedFeature :
  public cc_PMInspectFeature<ccPMInspectMatchedBP>;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a feature that is present in both the trained PatInspect pattern and the run-time image.

Note You should not instantiate this class directly.

Constructors/Destructors

ccPMInspectMatchedFeature

```
ccPMInspectMatchedFeature();
```

Construct an uninitialized **ccPMInspectMatchedFeature**.

Operators

operator==

```
bool operator== (const ccPMInspectMatchedFeature&) const;
```

Returns true if the location, direction, weight, match quality, and matched state are equal.

Public Member Functions

minDeformation

```
double minDeformation() const;
```

Returns the minimum deformation over all feature boundary points in this feature in alignment training image client coordinate system units. Feature boundary points in the run-time image that were not present in the inspection training image are ignored.

maxDeformation `double maxDeformation() const;`

Returns the maximum deformation over all feature boundary points in this feature in alignment training image client coordinate system units. Feature boundary points in the run-time image that were not present in the inspection training image are ignored.

ccPMInspectPattern

```
#include <ch_cvl/pminspct.h>

class ccPMInspectPattern : public cc_PMPattern;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class implements a PatInspect pattern. This is the class upon which the operation of the PatInspect vision tool is based. The two main components of a PatInspect pattern are:

- An alignment region that is used to align training and run-time inspection images
- An array of inspection regions that can be different from the alignment region and can be inspected according to the following inspection modes:

  intensity difference
  feature difference
  blank scene inspection

The member functions of **ccPMInspectPattern** allow you to:

- Train an inspection image
- Align an inspection image
- Run an inspection
- Return the results of the inspection

For more information on PatInspect training and run-time operation see *PatInspect* in *Vision Tools Guide*.

Notes

See the **cc_PMPattern** class for additional functions common to all PatMax pattern classes.

If you attempt to persist a **ccPMInspectPattern** before calling **endTrain()**, the archive operator will throw *ccPMInspectDefs::NotTrained*.

Constructors/Destructors

ccPMInspectPattern

```
ccPMInspectPattern();
```

Constructs an untrained **ccPMInspectPattern**.

Public Member Functions

compatibilityMode

```
void compatibilityMode(bool beCompatible);

bool compatibilityMode() const;
```

- `void compatibilityMode(bool beCompatible);`

  Causes this **ccPMInspectPattern** to be trained in CVL 5.4 backward compatibility mode. If *beCompatible* is set to true, all default values and behaviors are set according to CVL 5.4 conventions. The default values of *beCompatible* is false (the tool is not CVL 5.4 backward compatible by default).

  Parameters

  beCompatible If *true* PatInspect operates in CVL 5.4 backward compatibility mode. If *false* PatInspect is not CVL 5.4 backward compatible.

  Throws

  ccPMInspectDefs::BadParams
  The function is called after **ccPMInspectPattern::startTrain()**.

  Notes

  This function must be called only once (you cannot switch back and forth between compatibility modes) and immediately after the construction of this **ccPMInspectPattern**. Setting *beCompatible* to *true* prevents you from performing multiple region inspections.

- `bool compatibilityMode() const;`

  Returns true if this **ccPMInspectPattern** is configured to be CVL 5.4 backward compatible.

customizeInspect

```
ccCvlString customizeInspect(const ccCvlString& pcp);
```

Uses a Cognex-supplied customization string to modify internal PatInspect parameters. This function returns a string containing diagnostic information for use by Cognex Corporation.

Parameters

pcp A PatInspect customization string provided by Cognex Corporation.

Throws

cc_PMDefs::BadParams
This function is not supported in compatibility mode.

ccParamCompileError
pcp is invalid.

customizeInspectFromFile

```
ccCvlString customizeInspectFromFile(
  const ccCvlString& filename);
```

Uses a Cognex-supplied customization file to modify internal PatInspect parameters. This function returns a string containing diagnostic information for use by Cognex Corporation.

Parameters

filename A PatInspect customization file provided by Cognex Corporation.

Throws

cc_PMDefs::BadParams
This function is not supported in compatibility mode.

ccParamCompileError
filename contains invalid data or *filename* could not be opened.

customizeInspectString

```
ccCvlString customizeInspectString() const;
```

Returns the customization string most recently supplied to **customizeInspect()** or **customizeInspectFromFile()**.

Throws

cc_PMDefs::BadParams
This function is not supported in compatibility mode.

ignorePolarity

```
bool ignorePolarity() const;

void ignorePolarity(bool doIgnore);
```

- `bool ignorePolarity() const;`

  Returns true if this **ccPMInspectPattern** is configured to ignore polarity changes when searching for the pattern during alignment.

- `virtual void ignorePolarity(bool doIgnore);`

  Controls whether this **ccPMInspectPattern** is configured to ignore polarity changes when searching for the pattern during alignment. You can call this function without needing to retrain this **ccPMInspectPattern**. The default value of *doIgnore* is false.

  Parameters

doIgnore	If *true*, pattern polarity is ignored. If *false*, only patterns with matching polarity are found.

sobelCoeffs

```
void sobelCoeffs(const ccDPair& coeffs,
  c_Int32 index = 0);

ccDPair sobelCoeffs(c_Int32 index = 0) const;
```

- ```
  void sobelCoeffs(const ccDPair& coeffs,
    c_Int32 index = 0);
  ```

  Sets the coefficients used to compute the synthetic standard deviation image for the inspection region *index*. Each pixel in the standard deviation image is set to the following value:

  $$xM + y$$

  where:

  x is the value returned by *coeffs.x()*.

  M is the Sobel edge magnitude value of the pixel (see *Edge Tool* in *Vision Tools Guide* for more details).

  y is the value returned by *coeffs.y()*.

  The default values for x and y are 1.0 and 0.0. If you operate in CVL 5.4 backward compatibility mode the default values are 1.2 and 19.3.
  ```

Parameters

coeffs A **ccDPair** containing the x and y coefficients.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support intensity difference mode.

Notes

This function will have an effect only if **ccPMInspectPattern::endTrain()** is called in *ccPMInspectDefs::eIntensityDifference* mode with a single prior call to **ccPMInspectPattern::statisticTrain()**. You can call this function anytime before calling **ccPMInspectPattern::run()**. If you are working in CVL 5.4 backward compatibility mode and you want to set the values of x and y, you should call this function only after **ccPMInspectPattern::compatibilityMode(true)** has been called.

- ```
  ccDPair sobelCoeffs(c_Int32 index = 0) const;
  ```

Returns the coefficients used to compute the standard deviation region from the single inspection region *index*.

Parameters

index The index for the inspection region.

thresholdCoeffs

```
void thresholdCoeffs(const ccDPair& coeffs,
  c_Int32 index = 0);

ccDPair thresholdCoeffs(c_Int32 index = 0) const;
```

- ```
  void thresholdCoeffs(const ccDPair& coeffs,
    c_Int32 index = 0);
  ```

Sets the coefficients used to compute the threshold from the standard deviation for the inspection region *index*. Each pixel in the threshold region is set to the following value:

$$xS + y$$

where

x is the value returned by *coeffs.x()*.

S is the value of the pixel in the standard deviation region.

y is the value returned by *coeffs.y()*.

The default values for x and y are1.0 and 0.0. If you operate in CVL 5.4 backward compatibility mode the default values are 3.0 and 10.0.

Parameters

coeffs A **ccDPair** containing the x and y coefficients.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support intensity difference mode

Notes

You can call this function anytime before calling **ccPMInspectPattern::run()**. If you are working in CVL 5.4 backward compatibility mode and you want to set the values of x and y, you should call this function only after **ccPMInspectPattern::compatibilityMode(true)** has been called.

- `ccDPair thresholdCoeffs(c_Int32 index = 0) const;`

Returns the coefficients x and y used to compute the threshold image from the standard deviation image for the inspection region *index*.

Parameters

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support intensity difference mode.

tailFractions

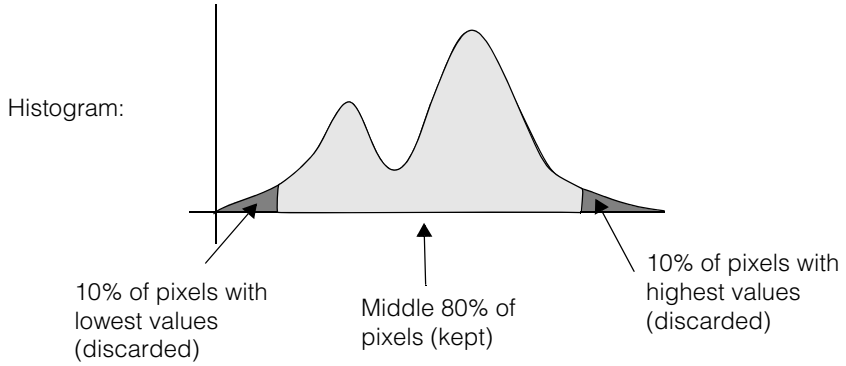
```
void tailFractions(const ccDPair& fracs,
  c_Int32 index = 0);

ccDPair tailFractions(c_Int32 index = 0) const;
```

- ```
  void tailFractions(const ccDPair& fracs,
    c_Int32 index = 0);
  ```

  Sets the tail percentages used for tail matching for the inspection region *index*. The fraction of tail pixels below the value supplied in *fracs.x()* and above the value supplied in *fracs.y()* are discarded before the run-time region is mapped. Legal values for *fracs.x()* and *fracs.y()* are between 0.0 and 1.0.

  The following figure shows how the tail fractions are interpreted, given fractions of 0.10 and 0.90:



  The default values for the left and right tail values are 0.1 and 0.9.

  Parameters

  fracs — A **ccDPair** containing the left and right tail lengths. Whichever value is higher is used as the right tail length.

  index — The index for the inspection region.

  Throws

  ccPMInspectDefs::BadParams

  fracs contains values less than 0.0 or greater than 1.0.
  The index for the inspection region is not valid.
  Region *index* does not support intensity difference mode.

  Notes

  You can call this function anytime before calling **ccPMInspectPattern::run()**.
  ```

```
ccDPair tailFractions(c_Int32 index = 0) const;
```

Returns the tail percentages used for tail matching for the inspection region *index*.

Throws

ccPMInspectDefs::BadParams

fracs contains values less than 0.0 or greater than 1.0.
The index for the inspection region is not valid.
Region *index* does not support intensity difference mode.

interpolationQuality

```
void interpolationQuality(
  ccAffineSamplingParams::Interpolation value,
  c_Int32 index = 0);

ccAffineSamplingParams::Interpolation
  interpolationQuality(c_Int32 index = 0) const;
```

- ```
  void interpolationQuality(
    ccAffineSamplingParams::Interpolation value,
    c_Int32 index = 0);
  ```

Sets the interpolation method used to affine-transform the training and run-time inspection region *index*.

Parameters

value — A valid sampling method as defined in *ccAffineSamplingParams::Interpolation*.

index — The index for the inspection region.

Throws

ccPMInspectDefs::BadParams

value is not one of the enumerated values defined in *ccAffineSamplingParams::Interpolation*.
index is not valid for the inspection region.

Notes

Interpolation quality is relevant for intensity difference and blank scene modes and does not affect boundary difference mode.

This function must be called before **ccPMInspectPattern::statisticTrain()**.
  ```

- ```
  ccAffineSamplingParams::Interpolation
    interpolationQuality(c_Int32 index = 0) const;
  ```

  Returns the sampling method used to affine-transform the run-time inspection region *index* before computing the intensity difference image.

  Throws

  ccPMInspectDefs::BadParams
  index is not valid for the inspection region.

  Notes

  Interpolation quality is relevant for intensity difference and blank scene modes and does not affect boundary difference mode.

templateImage

```
const ccPelBuffer_const<c_UInt8>& templateImage(
  c_Int32 index = 0) const;
```

Returns the template region used for intensity difference inspection of the region *index*.

Parameters

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support intensity difference mode.

Notes

This function returns an unbound image if this **ccPMInspectPattern** has not completed statistical training for intensity difference mode.

thresholdImage

```
const ccPelBuffer_const<c_UInt8>& thresholdImage(c_Int32
  index = 0) const;
```

Returns the threshold image used for intensity difference inspection of the region *index*.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support intensity difference mode.

Notes

This function returns an unbound image if this **ccPMInspectPattern** has not completed statistical training for intensity difference mode.
```

maskImage `const ccPelBuffer_const<c_UInt8>& maskImage() const;`

Returns the mask image used for alignment and for inspection region 0 when operating in CVL 5.4 backward compatibility mode.

Notes

This function returns an unbound image if training of this **ccPMInspectPattern** has not started or if a mask image was not supplied.

ignoreInspectPolarity

```
void ignoreInspectPolarity(bool doIgnore, c_Int32 index);
bool ignoreInspectPolarity(c_Int32 index);
```

- `void ignoreInspectPolarity(bool doIgnore, c_Int32 index);`

Controls whether to ignore the polarity of the inspection region *index* when inspecting in feature difference mode.

Parameters

doIgnore If *true*, the polarity of the region *index* is ignored during feature difference inspection. If *false*, polarity is taken into account.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

- `bool ignoreInspectPolarity(c_Int32 index);`

Returns true if the polarity of the inspection region *index* is ignored when inspecting in feature difference mode.

Parameters

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

matchQualityThresholds

```
void matchQualityThresholds(double low, double high,
  c_Int32 index = 0);
```

Sets the low and high thresholds for match quality for the inspection region *index*. Only features with average match quality greater than the high threshold are treated as matching features. Features with match quality between the low and high thresholds are discarded. Features with match quality below the low threshold are treated as either missing or extraneous.

If the two thresholds are equal (which is the default), then no features are discarded, and all features appear in one of the three feature lists (matched, missing, or extra).

Valid values are in the range -1.0 through 1.0. The default values are 0.0 for both the low threshold and the high threshold.

Parameters

low The low threshold.

high The high threshold.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams

low is greater than *high*.
The inspection region index is not valid.
Region *index* does not support feature difference mode.

matchQualityThresholdLow

```
double matchQualityThresholdLow(c_Int32 index = 0) const;
```

Returns the match quality low threshold assigned to the inspection region *index*.

Parameters

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams

The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

matchQualityThresholdHigh

```
double matchQualityThresholdHigh(c_Int32 index = 0) const;
```

Returns the match quality high threshold of the inspection region *index*.

Parameters

index The index for the inspection region

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

boundaryDeformation

```
void boundaryDeformation(double value, c_Int32 index);

double boundaryDeformation(c_Int32 index) const;
```

- `void boundaryDeformation(double value, c_Int32 index);`

Sets the distance in pixels within which feature boundary points can be matched for the inspection region *index*. This parameter indicates how far away a feature boundary point in the run-time region can be from a feature boundary point in the template region and still be considered for a match. The default value is 0.0. The maximum value is 5.0.

Parameters

value The distance in pixels within which feature boundary points can be matched.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
value is less than 0.0 or greater than 5.0.
The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

Notes

Calling this function has no effect in this release.

- `double boundaryDeformation(c_Int32 index) const;`

Returns the distance in pixels within which feature boundary points can be matched for the inspection region *index*.

Parameters

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

Notes
Calling this function has no effect in this release.

background

```
void background(double back, c_Int32 index);
double background(c_Int32 index) const;
```

- `void background(double back, c_Int32 index);`

Overrides the standard deviation value computed during blank scene training of the region *index* and sets it to *back* (see *Training for Blank Scene Inspection* in the *Vision Tools Guide* for a description of how the standard deviation value is computed during blank scene training).

Parameters
back The value the standard deviation region is set to.

index The index for the inspection region.

Throws
ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support blank scene inspection mode.

Notes
The value of *back* is automatically generated during training but you can use this function to override it.

- `double background(c_Int32 index) const;`

Returns the standard deviation value used for blank scene inspection training of the region *index*.

Parameters
index The index for the inspection region.

Throws
ccPMInspectDefs::BadParams
The index for the inspection region is not valid.
Region *index* does not support blank scene inspection mode.

backgroundRange

```
void backgroundRange(double backRange, c_Int32 index);
double backgroundRange(c_Int32 index) const;
```

- `void backgroundRange(double backRange, c_Int32 index);`

  Sets the coefficient used to compute the blank scene threshold from the standard deviation of the region *index (*see *Training for Blank Scene Inspection* in the *Vision Tools Guide* for a description of how the threshold region is obtained during blank scene training). Each pixel in the threshold region is set to the following value:

  $$backRange \times S$$

  where S is the standard deviation value of the region *index*. The value of *backRange* must be from 0.0 through 5.0.

  Parameters

  backRange — The coefficient used to compute the threshold for blank scene inspection of region *index*.

  index — The index for the inspection region.

  Throws

  ccPMInspectDefs::BadParams

  backRange is either less than 0.0 or greater than 5.0.
  The index for the inspection region is not valid.
  Region *index* does not support blank scene inspection mode.

- `double backgroundRange(c_Int32 index) const;`

  Returns the coefficient used to compute the blank scene threshold from the standard deviation of the region *index.*

  Parameters

  index — The index for the inspection region.

  Throws

  ccPMInspectDefs::BadParams

  The index for the inspection region is not valid.

minimumFeatureSize

```
void minimumFeatureSize(c_Int32 minSize, c_Int32 index);

c_Int32 minimumFeatureSize(c_Int32 index) const;
```

- `void minimumFeatureSize(c_Int32 minSize, c_Int32 index);`

  Sets the minimum boundary feature size in pixels for the region *index*. Features with sizes that are less than *minSize* are not considered defects. The default value is 1.0.

  Throws

  ccPMInspectDefs::BadParams

  The index for the inspection region is not valid.
  Region *index* does not support either feature difference or blank scene inspection mode.

  Notes

  This function can be used for both blank scene and feature difference inspection mode.

- `c_Int32 minimumFeatureSize(c_Int32 index) const;`

  Returns the minimum boundary feature size in pixels for the region *index*.

  Throws

  ccPMInspectDefs::BadParams

  The index for the inspection region is not valid.
  Region *index* does not support either feature difference or blank scene inspection mode.

  Notes

  This function can be used for both blank scene and feature difference inspection mode.

minimumFeatureContrast

```
void minimumFeatureContrast(double contrastThresh,
  c_Int32 index);

double minimumFeatureContrast(c_Int32 index) const;
```

- `void minimumFeatureContrast(double contrastThresh, c_Int32 index);`

  Sets the minimum feature contrast in grey levels for the region *index*. Features with contrast levels that are less than *contrastThresh* are not considered defects. The default value of *contrastThresh* is 10.0 grey levels.

Throws

ccPMInspectDefs::BadParams

contrastThresh is either less than 0 or greater than 255
The index for the inspection region is not valid.
Region *index* does not support either feature difference or blank scene inspection mode.

Notes

This function can be used for both blank scene and feature difference inspection mode.

- ```
  void minimumFeatureContrast(double contrastThresh,
    c_Int32 index);
  ```

Returns the minimum feature contrast in grey levels for the region *index.*

Throws

ccPMInspectDefs::BadParams

The index for the inspection region is not valid.
Region *index* does not support either feature difference or blank scene inspection mode.

Notes

This function can be used for both blank scene and feature difference inspection mode.

trainFeatureContrast

```
void trainFeatureContrast(double contrastThresh,
  c_Int32 index);

double trainFeatureContrast(c_Int32 index) const;
```

- ```
  void trainFeatureContrast(double contrastThresh,
    c_Int32 index);
  ```

Sets the minimum feature contrast in grey levels for the region *index* at training time. Features with contrast levels that are less than *contrastThresh* are not included in the trained **ccPMInspectPattern**. The default value of *contrastThresh* is 10.0 grey levels.

Throws

ccPMInspectDefs::BadParams

contrastThresh is either less than 0 or greater than 255; the index for the inspection region is not valid; or the region *index* does not support either feature difference or blank scene inspection mode.

Notes

This function can be used for both blank scene and feature difference inspection mode.

- ```
  void trainFeatureContrast(double contrastThresh,
    c_Int32 index);
  ```

Returns the minimum feature contrast in grey levels for the region *index* at training time. Features with contrast levels that are less than *contrastThresh* are included in the trained **ccPMInspectPattern**. The default value of *contrastThresh* is 10.0 grey levels.

Throws

ccPMInspectDefs::BadParams

The index for the inspection region is not valid or region *index* does not support either feature difference or blank scene inspection mode.

Notes

This function can be used for both blank scene and feature difference inspection mode.

---

startTrain

```
void startTrain(ccDiagObject* dObj=0,
  c_UInt32 diagFlags=0);

void startTrain(
  const ccPelBuffer_const<c_UInt8>& alignModel,
  ccDiagObject* dObj=0, c_UInt32 diagFlags=0);

void startTrain(
  const ccPelBuffer_const<c_UInt8>& alignModel,
  const ccPelBuffer_const<c_UInt8>& mask
  ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
```

- ```
  void startTrain(ccDiagObject* dObj=0,
    c_UInt32 diagFlags=0);
  ```

Starts training when alignment is performed by using a pose. You supply the pose when you call the appropriate overload of **ccPMInspectPattern::statisticTrain()**.

Parameters

dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMInspectDefs::BadParams
PatInspect operates in backward compatibility mode.

ccPel::BadWindow
Any inspection region defined with the **ccAffineRectangle** overload of **addInspectRegion()** is out of the image.

Notes

To avoid *ccPel::BadWindow* being thrown from **startTrain()** when using affine rectangular regions that are clipped by the root image, test for a bad **ccAffineRectangle** before calling **addInspectRegion()**. For example, the following code performs such a test:

```
ccAffineRectangle affRect(cc2Xform(cc2Vect(267,123),
  ccRadian(ccDegree(-27)), ccRadian(ccDegree(-27)), 300,
  150));
// Test affine rectangle before calling addInspectRegion()
ccPelBuffer <c_UInt8> foo(trainImage);
try
{
  foo.window(affRect.encloseImageRect());
}
catch(ccPel::BadWindow &e)
{
  // bad ccAffineRectangle, try again!
  throw;
}
pat.addInspectRegion(model, ccPMInspectDefs::eBPDifference,
  cmT("BFC"));
```

See **ccPMInspectPattern::addInspectRegion()** for more information.

- ```
  void startTrain(
    const ccPelBuffer_const<c_UInt8>& alignModel,
    ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
  ```

  Performs alignment training using the alignment region *alignModel*. When operating in backward compatibility mode, *alignModel* is also inspection region 0. PatInspect uses the pattern information trained from *alignModel* to locate the alignment region in both training and run-time images. All previously trained information is discarded after the call to this function.

  Parameters

  alignModel — The region to use for alignment training.

  dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccPel::BadWindow

  Any inspection region defined with the **ccAffineRectangle** overload of **addInspectRegion()** is out of the image. See notes in first overload for details.
  ```

- ```
  void startTrain(
    const ccPelBuffer_const<c_UInt8>& alignModel,
    const ccPelBuffer_const<c_UInt8>& mask,
    ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
  ```

  Performs alignment training using the alignment region *alignModel* and the masking image *mask*.

  The mask image must have the same dimensions and offset as the alignment region. The pixels in the mask image are interpreted as follows:

  - All pixels in *alignModel* that correspond to pixels in *mask* with values greater than or equal to 192 are considered 'care' pixels. All feature boundary points detected within 'care' pixels are included in the trained pattern.
  - All pixels in *alignModel* that correspond to pixels in *mask* with values from 0 through 63 are considered 'don't care but score' pixels. Feature boundary points detected within 'don't care but score' pixels are not included in the trained pattern.

    When the trained pattern is located in a run-time image, features within the 'don't care but score' part of the trained pattern are treated as clutter features and they contribute to the feature difference list.
  - All pixels in *alignModel* that correspond to pixels in *mask* with values from 64 through 127 are considered 'don't care and don't score' pixels. Feature boundary points detected within 'don't care and don't score' pixels are not included in the trained pattern.

    When the trained pattern is located in a run-time image, features within the 'don't care and don't score' part of the trained pattern are ignored and not treated as clutter features and they do not contribute to the feature difference list.
  - Mask pixel values from 128 through 191 are reserved for future use by Cognex.

  Parameters

alignModel	The region to be used for alignment training.
mask	The mask image. *mask* must have the same dimensions and offset as *alignModel*. The client coordinate system of *mask* is ignored.
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults*
  ```

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

All results produced before this call become invalid. *alignModel* is used for alignment training only.

Throws

ccPMInspectDefs::CanNotTrain

The pattern could not be trained or *mask* does not have the same dimensions and offset as *alignModel*.

ccPMInspectDefs::BadImage

PatInspect could not locate a pattern instance with a score greater than the acceptance threshold within the image.

ccPel::BadWindow

Any inspection region defined with the **ccAffineRectangle** overload of **addInspectRegion()** is out of the image. See notes in first overload for details.

statisticTrain

```
cc2Xform statisticTrain(
  const ccPelBuffer_const<c_UInt8>& trainScene,
  const ccPMInspectStatTrainParams& params,
  c_Int32 inspectModes =
  ccPMInspectDefs::eIntensityDifference,
  ccDiagObject* dObj=0, c_UInt32 diagFlags=0);

cc2Xform statisticTrain(
  const ccPelBuffer_const<c_UInt8>& trainScene,
  const ccPMInspectStatTrainParams& params,
  const cc2Xform& pose,
  c_Int32 inspectModes =
  ccPMInspectDefs::eIntensityDifference,
  ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
```

-
```
cc2Xform statisticTrain(
  const ccPelBuffer_const<c_UInt8>& trainScene,
  const ccPMInspectStatTrainParams& params,
  c_Int32 inspectModes =
  ccPMInspectDefs::eIntensityDifference,
  ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
```

Adds an iteration to the statistical training of all the enabled regions in *trainScene* and returns the pose at which the alignment region was found. The parameters controlling the search of the alignment region are defined in *params*. If you are in CVL 5.4 backward

compatibility mode, the *inspectModes* parameter is applied to inspection region 0. If you are not in backward compatibility mode the value of *inspectModes* is ignored and need not be supplied.

Parameters

trainScene The image to use for inspection training.

params A **ccPMInspectStatTrainParams** object containing the parameters used to locate the alignment region in the training image.

inspectModes The inspection mode to be used for inspection region 0 when in backward compatibility mode. *inspectModes* must be formed by ORing together one or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference

If you supply 0 for *inspectModes*, the function returns **cc2Xform::I()** and performs no training.

dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMInspectDefs::NotTrained
The **ccPMInspectPattern::startTrain()** function has not been called on this **ccPMInspectPattern**.

ccPMInspectDefs::BadImage
PatInspect cannot locate the alignment region within the training image with a score greater than the acceptance threshold.
The alignment region is not fully in the field if view.
One or more regions to be trained for blank scene inspection contain too many edge features to be considered blank.

ccPMInspectDefs::CanNotTrain
ccPMInspectPattern::endTrain() has been called with the *saveState* parameter set to *false;* the no argument overload of **ccPMInspect::startTrain()** has been called; or one or more of the regions to be trained for boundary mode has too many features.

Notes

You must have already called the appropriate overload of **ccPMInspect::startTrain()** before calling this function, and you must call **ccPMInspect::statisticTrain()** at least once between calls to **ccPMInspect::startTrain()** and **ccPMInspect::endTrain()** for correct operation. *trainScene* must span all the window regions supplied as arguments to **addInspectRegion()**. If you are not in backward compatibility mode the value of *inspectMode* is ignored. If a throw occurs, training terminates for all inspection regions regardless of the region that caused the throw.

- ```
  cc2Xform statisticTrain(
    const ccPelBuffer_const<c_UInt8>& trainScene,
    const ccPMInspectStatTrainParams& params,
    const cc2Xform& pose,
    c_Int32 inspectModes =
    ccPMInspectDefs::eIntensityDifference,
    ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
  ```

  Adds an iteration to the statistical training of all the enabled regions in *trainScene. trainScene* is aligned by using *pose* and the function returns *pose*. If you are in CVL 5.4 backward compatibility mode, the *inspectModes* parameter is applied to inspection region 0. If you are not in backward compatibility mode the value of *inspectModes* is ignored.

  Parameters

  trainScene — The image to use for inspection training.

  params — A **ccPMInspectStatTrainParams** object. This parameter is a place holder reserved for future use by Cognex. The value of this parameter is ignored in this release.

  pose — The transformation used to align *trainScene*.

  inspectModes — The inspection mode to be used for inspection region 0 when in backward compatibility mode. *inspectModes* must be formed by ORing together one or more of the following values:

  ccPMInspectDefs::eIntensityDifference
  ccPMInspectDefs::eBPDifference
  ```

If you supply 0 for *inspectModes*, the function returns **cc2Xform::I()**, and it performs no training.

dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMInspectDefs::NotTrained
The **startTrain()** function has not been called on this **ccPMInspectPattern**.

ccPMInspectDefs::BadImage
One or more regions to be trained for blank scene inspection contain too many edge features to be considered blank.

ccPMInspectDefs::CanNotTrain
ccPMInspectPattern::endTrain() has been called with the *saveState* parameter set to *false;* the no argument overload of **ccPMInspect::startTrain()** has been called; or one or more of the regions to be trained for boundary mode has too many features.

Notes

You must have already called the appropriate overload of **ccPMInspect::startTrain()** before calling this function, and you must call **ccPMInspect::statisticTrain()** at least once between calls to **ccPMInspect::startTrain()** and **ccPMInspect::endTrain()** for correct operation. *trainScene* must span all the window regions supplied as arguments to **addInspectRegion()**.

endTrain

```
void endTrain(bool saveState,
  c_Int32 inspectModes =
  ccPMInspectDefs::eIntensityDifference,
  ccDiagObject* dObj=0, c_UInt32 diagFlags=0);
```

Completes the training process for this **ccPMInspectPattern**. If *saveState* is true then intermediate state information is saved and you can call **ccPMInspect::statisticTrain()** later to add more training images. If *saveState* is false, then the memory taken up by the state is freed, but the ability to further train the pattern is lost. If you are in CVL 5.4 backward compatibility mode, the *inspectModes* parameter is applied to inspection region 0. If you are not in backward compatibility mode the value of *inspectModes* is ignored.

Parameters

saveState — If *true*, then internal state information is saved. If *false*, then internal state information is discarded.

inspectModes — The inspection mode of inspection region 0 when in backward compatibility mode. *inspectModes* must be formed by ORing together one or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference

dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccPMInspectDefs::NotTrained
ccPMInspect::startTrain() has not been called.
One of the difference patterns has not been trained.
The pattern has not been previously trained with the specified method.

ccPMInspectDefs::CanNotTrain
ccPMInspect::endTrain() has been previously called with *saveState* set to false or an inspection region using boundary

feature comparison mode is not trainable. Note that you can disable an individual untrainable region without discarding all regions.

Notes

For blank scene inspection mode this function is only meant as a mechanism to recompute and update internal statistical information. When in this inspection mode, the value of *saveState* has no effect and you can continue training even after the function is called.

untrain

```
void untrain(bool alignOnly = false);
```

Frees the memory allocated for the internal data. If *alignOnly* is true only the memory allocated for the alignment pattern is freed. If *alignOnly* is false the memory allocated for the alignment pattern and the inspection regions is freed. This function can be called on an untrained pattern with no effect.

Parameters

alignOnly — If *true*, then the memory allocated for the alignment pattern is freed. If *false*, the memory for the inspection regions is also freed.

Notes

All the results produced before the call to this function became invalid.

addInspectRegion

```
c_Int32 addInspectRegion(
  const ccPelBuffer_const<c_UInt8>& inspReg,
  c_UInt32 inspectModes, const ccCvlString& name = cmT(""));

c_Int32 addInspectRegion(const ccAffineRectangle& inspReg,
  c_UInt32 inspectModes, const ccCvlString& name = cmT(""));
```

-
```
c_Int32 addInspectRegion(
  const ccPelBuffer_const<c_UInt8>& inspReg,
  c_UInt32 inspectModes, const ccCvlString& name = cmT(""));
```

Adds the inspection region *inspReg* to the current inspection region list and returns the total number of regions after the addition. The region is added in the enabled state and is identified by the case-sensitive string *name*. *inspectModes* specifies the inspection mode of *inspReg, a combination of the* enums *ccPMInspectDefs::InspectMode* and *ccPMInspectDefs::InspectModeModifier*. If *inspReg* is inspected for blankness, it cannot be also inspected for intensity or feature difference. The inspection modifiers

apply only to feature difference modes. The combinations of inspection base modes and modifiers that can be applied to an inspection region are summarized in the following table.

Base Mode	Valid additional base mode	Modifiers
eBPDifference	*eIntensityDifference*	*eSoftwareZoomTrain* *eSoftwareZoomTrainRun* *eFineAlign*
eIntensityDifference	*eBPDifference*	*eSoftwareZoomTrain* *eSoftwareZoomTrainRun* *eFineAlign*
eBlankScene	None	None

This function must be called before **ccPMInspect::startTrain()**.

Parameters

inspReg The region added to the list of inspection regions.

inspectModes The inspection mode of *inspReg*. This parameter must be formed by ORing together valid combinations of the base modes and modifiers as show in the table above.

name An optional string that identifies *inspReg*.

Throws

ccPMInspectDefs::NotTrained
ccPMInspect::startTrain() has been called first.

ccPMInspectDefs::BadParams
The region could not be added to the list.
A region with the same name already exists.
An invalid bit is asserted in *inspectModes*.
PatInspect operates in CVL 5.4 backward compatibility mode.

- ```
  c_Int32 addInspectRegion(const ccAffineRectangle& inspReg,
    c_UInt32 inspectModes, const ccCvlString& name = cmT(""));
  ```

  Same as above, except it adds the inspection region specified by the supplied **ccAffineRectangle** to the current inspection region list.

  Parameters

  inspReg The **ccAffineRectangle** added to the list of inspection regions.
  ```

inspectModes The inspection mode of *inspReg*. This parameter must be formed by ORing together valid combinations of the base modes and modifiers as show in the table above.

name An optional string that identifies *inspReg.*

Throws

ccPMInspectDefs::NotTrained
ccPMInspect::startTrain() has been called first.

ccPMInspectDefs::BadParams
The region could not be added to the list; a region with the same name already exists; an invalid bit is asserted in *inspectModes.*; PatInspect is being operated in CVL 5.4 backward compatibility mode; or *inspReg* specifies a non-zero skew component.

inspectRegion

```
const ccPMInspectRegion& inspectRegion(
  c_Int32 index)const;

ccPMInspectRegion& inspectRegion(c_Int32 index);
```

- ```
  const ccPMInspectRegion& inspectRegion(
    c_Int32 index) const;
  ```

  Returns a read-only reference to the inspection region *index*.

  Parameters

  index The index for the inspection region.

- ```
  ccPMInspectRegion& inspectRegion(c_Int32 index);
  ```

  Returns a reference to the inspection region *index*.

  Parameters

  index The index for the inspection region.

  Throws

  ccPMInspectDefs::BadParams
  The index for the inspection region is not valid.

inspectRegionIndex

```
c_Int32 inspectRegionIndex(const ccCvlString& name) const;
```

Returns the index for the inspection region identified by the string *name.*

Parameters

name The string that identifies the inspection region

Throws

ccPMInspectDefs::BadParams
There is no region identified by *name.*

numInspectRegions

```
c_Int32 numInspectRegions(c_UInt32 mask = 0,
  bool onlyIfEnabled = false) const;
```

Returns the total number of inspection regions whose inspection mode is *mask*. If *onlyIfEnabled* is true only the regions enabled for inspection are counted.

Parameters

mask The inspection mode. This parameter must be formed by ORing together one or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference
ccPMInspectDefs::InspectModeModifier

You can assign *cPMInspectDefs::eBlankScene* to *inspReg* but you cannot OR *cPMInspectDefs::eBlankScene* with any of the previous enumerations.

onlyIfEnabled If *true* only the enabled regions are counted. If *false* all the regions are counted.

Throws

ccPMInspectDefs::BadParams
The inspection modality specified by *mask* is not valid.

inspectRegionType

```
void inspectRegionType(cmStd vector<c_UInt32>& codes,
  bool enableMode=false) const;
```

Resizes and initializes the vector *codes*. Each element of the vector contains the bitwise OR of values from the *ccPMInspectDefs::InspectMode* and *ccPMInspectDefs::InspectModeModifier* enumerations. If *enableMode* is true then the vector elements corresponding to the inspection regions that are not enabled are set to zero.

Parameters

codes The vector containing the information on the inspection mode of each region.

enableMode If *true*, the vector elements corresponding to the inspection regions that are not enabled are set to zero. If *false,* the inspection mode of each region is reported.

enableAllRegions

```
void enableAllRegions();
```

Enables all inspection regions.

disableAllRegions

```
void disableAllRegions();
```

Disables all inspection regions.

displayInspectConfig

```
void displayInspectConfig(ccGraphicList& displayList)
 const;
```

Initializes the specified *displayList* with a graphical and text rendering of the current inspection configuration using default colors.

Parameters

displayList The graphic list used to display the current inspection configuration.

Notes

Before training, this function returns the region windows and annotations configured for this pattern. After training, this function returns the region windows and annotations as well as the trained model feature graphics. Note that incomplete results will be obtained when displaying inspection regions configured in boundary feature mode if this function is called before training has completed.

timesStatTrained

```
c_Int32 timesStatTrained(c_UInt32 mask, c_Int32 index)
 const;
```

Returns the number of iterations of statistical training that were performed for the inspection region *index* in the inspection mode specified by *mask*. Returns 0 if the alignment region has not been trained.

Parameters

mask The inspection mode. This parameter must be formed by ORing together 0 or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference

ccPMInspectDefs::InspectModeModifier

You can assign *cPMInspectDefs::eBlankScene* to *inspReg* but you cannot OR *cPMInspectDefs::eBlankScene* with any of the previous enumerations.

index The index for the inspection region

Throws

ccPMInspectDefs::BadParams
The inspection mode specified by *mask* is not valid.
The index for the inspection region is not valid

isModeTrained

```
bool isModeTrained(ccPMInspectDefs::InspectMode mode,
 c_Int32 index) const;

bool isModeTrained(ccPMInspectDefs::InspectMode mode,
 const ccCvlString& name) const;
```

- ```
  bool isModeTrained(ccPMInspectDefs::InspectMode mode,
   c_Int32 index) const
  ```

Returns true if the inspection region *index* has completed training in the inspection mode specified by *mode*.

Parameters

mode The inspection mode. This parameter must be formed by ORing together one or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference
ccPMInspectDefs::eBlankScene

Note: When you set an inspection region to *ccPMInspectDefs::eBlankScene* you cannot OR this mode with any of the other modes. See *addInspectRegion* on page 2518 for valid combinations. The mode modifiers shown there are not required for this function.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams
The inspection modality specified by *mode* is not valid.
The index for the inspection region is not valid.
  ```

- `bool isModeTrained(ccPMInspectDefs::InspectMode mode, const ccCvlString& name) const;`

  Returns true if the inspection region identified by *name* has completed training in the inspection mode specified by *mode*.

  Parameters

  mode The inspection mode. This parameter must be formed by ORing together one or more of the following values:

  ccPMInspectDefs::eIntensityDifference
  ccPMInspectDefs::eBPDifference
  ccPMInspectDefs::eBlankScene

  Note: When you set an inspection region to *ccPMInspectDefs::eBlankScene* you cannot OR this mode with any of the other modes. See *addInspectRegion* on page 2518 for valid combinations. The mode modifiers shown there are not required for this function.

  name The string that identifies the inspection region

  Throws

  ccPMInspectDefs::BadParams
  The inspection modality specified by *mode* is not valid. There is no inspection region identified by *name*.

isAlignmentTrained

```
bool isAlignmentTrained() const;
```

Returns true if the training of the alignment region is completed, false otherwise.

run

```
void run(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPMInspectRunParams& params,
  ccPMInspectResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 diagFlags=0);

void run(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPMInspectRunParams& params,
```

```
const cmStd vector<cc2Xform>& poses,
ccPMInspectResultSet& resultSet, ccDiagObject* dObj=0,
c_UInt32 diagFlags=0);
```

- ```
  void run(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPMInspectRunParams& params,
  ccPMInspectResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 diagFlags=0);
  ```

  Performs the inspection on *image* using the parameters specified by *params* and stores the inspection results in *resultSet*. When this function is called all the enabled regions are inspected according to their inspection mode.

  Parameters

  image — The image to inspect.

  params — A **ccPMInspectRunParams** object containing the inspection parameters.

  resultSet — A **ccPMInspectResultSet** object into which the inspection results are placed.

  dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccPMInspectDefs::NotTrained
  This **ccPMInspectPattern** is not trained.
  This **ccPMInspectPattern** was trained without an alignment region.

  ccPMInspectDefs::BadImage
  The inspection image is unbound.
  ```

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
    const ccPMInspectRunParams& params,
    const cmStd vector<cc2Xform>& poses,
    ccPMInspectResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 diagFlags=0);
  ```

  Performs the inspection on *image* using the parameters specified by *params* and stores the inspection results in *resultSet*. The function uses the vector of poses to perform region alignment. When this function is called all the enabled regions are inspected according to their inspection mode.

  Parameters

  image — The image to inspect.

  params — A **ccPMInspectRunParams** containing the inspection parameters.

  poses — A vector of **cc2Xform**s containing the poses to be used for aligning the inspection regions.

  *result*Set — A **ccPMInspectResultSet** object into which the inspection results are placed

  dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:

  ccDiagDefs::eRecordOn
  ccDiagDefs::eRecordOff

  Throws

  ccPMInspectDefs::NotTrained
  This **ccPMInspectPattern** is not trained.

  ccPMInspectDefs::BadImage
  The inspection image is unbound.

  Notes

  The length of the poses vector argument determines the number of results returned.
  ```

Deprecated Members

The following member functions are deprecated in CVL 5.5.1. They are provided for backward compatibility only and will not cause compilation errors in CVL 5.5.1. It is recommended to update your code to use the newer versions. These functions will be obsoleted after CVL 5.5.1, at which time they will cause compilation errors.

deformation

```
void deformation(double value);

double deformation() const;
```

```
void deformation(double value);
```

Sets the distance in pixels within which boundary points can be matched. This parameter indicates how far away a boundary point in the run-time region can be from a boundary point in the template region and still be considered for a match. The default value is 0.0. The maximum value is 5.0. This function applies only to inspection region 0 when in CVL 5.4 backward compatibility mode.

Parameters

value The distance in pixels within which image and pattern boundary points can be matched.

Throws

ccPMInspectDefs::BadParams

value is less than 0 or greater than 5.0
The index for the inspection region is not valid

- `double deformation() const;`

Returns the distance in pixels within which boundary points can be matched. This function applies only to inspection region 0 when in CVL 5.4 backward compatibility mode.

Notes

Calling this function has no effect.

minFeatureContrast

```
void minFeatureContrast(double contrastThresh);

double minFeatureContrast() const;
```

- `void minFeatureContrast(double contrastThresh);`

  Sets the minimum feature contrast. Only feature boundary points in the run-time image with contrast greater than or equal to the minimum feature contrast are considered as defects. Feature boundary points which do not have the minimum contrast level are discarded before the features are assembled. This function applies only to inspection region 0 when in CVL 5.4 backward compatibility mode.

  Parameters

  contrastThresh The minimum feature contrast, expressed as the number of grey levels.

  Throws

  ccPMInspectDefs::BadParams

  contrast is greater than 255.0 or less than 0.0.
  The index for the inspection region is not valid.

  Notes

  The default minimum feature contrast is 10.0.

- `double minFeatureContrast() const;`

  Returns the minimum feature contrast expressed in grey levels. This function applies only to inspection region 0 when in CVL 5.4 backward compatibility mode.

minFeatureSize

```
void minFeatureSize(c_Int32 minSize);

c_Int32 minFeatureSize() const;
```

- `void minFeatureSize(c_Int32 minSize);`

  Sets the minimum boundary feature size in pixels. This function applies only to inspection region 0.

- `c_Int32 minFeatureSize() const;`

  Returns the minimum boundary feature size in pixels. This function applies only to inspection region 0.

timesStatTrained

```
int timesStatTrained(
  ccPMInspectDefs::InspectMode mode) const;
```

Returns the number of iterations of statistical training performed on the inspection region 0. The inspection mode is defined by *mode*. This function applies only to inspection region 0 when in CVL 5.4 backward compatibility mode.

Parameters

mode — The inspection mode. This parameter must be formed by ORing together one or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference

You cannot OR *ccPMInspectDefs::eBlankScene* with any of the previous enumerations.

isDifferenceTrained

```
bool isDifferenceTrained(
  ccPMInspectDefs::InspectMode mode) const;
```

Returns true if this **ccPMInspectPattern** can be used to perform inspections of the specified inspection mode. This is determined by whether or not **endTrain()** has been successfully called for the specified difference type.

Parameters

mode — The inspection mode. This parameter must be formed by ORing together one or more of the following values:

ccPMInspectDefs::eIntensityDifference
ccPMInspectDefs::eBPDifference

Throws

ccPMInspectDefs::BadParams
The inspection modality specified by *mode* is not valid.

run

```
void run(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPMInspectRunParams& params,
  cmStd vector<ccPMInspectResult>& results);

void run(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPMInspectRunParams& params,
  const cmStd vector<cc2Xform>& poses,
  cmStd vector<ccPMInspectResult>& results);
```

-
```
void run(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccPMInspectRunParams& params,
  cmStd vector<ccPMInspectResult>& results);
```

Performs the inspection on *image* using the parameters specified by *params* and stores the inspection results in *results*. When this function is called all the enabled regions are inspected according to their inspection mode.

Parameters

image — The image to inspect.

params — A **ccPMInspectRunParams** object containing the inspection parameters.

results — A vector of **ccPMInspectResult** into which the inspection results are placed.

Throws

ccPMInspectDefs::NotTrained
This **ccPMInspectPattern** is not trained.
This **ccPMInspectPattern** was trained without an alignment region.

ccPMInspectDefs::BadImage
The inspection image is unbound.

- ```
  void run(
    const ccPelBuffer_const<c_UInt8>& image,
    const ccPMInspectRunParams& params,
    const cmStd vector<cc2Xform>& poses,
    cmStd vector<ccPMInspectResult>& results);
  ```

  Performs the inspection on *image* using the parameters specified by *params* and stores the inspection results in *results*. The function uses the vector of poses to perform region alignment. When this function is called all the enabled regions are inspected according to their inspection mode.

  Parameters

  image — The image to inspect.

  params — A **ccPMInspectRunParams** object containing the inspection parameters.

  poses — A vector of **cc2Xform**s containing the poses to be used for aligning the inspection regions.

  results — A vector of **ccPMInspectResult**s into which the inspection results are placed.

  Notes

  The length of the *poses* vector argument determines the number of results returned.

  Throws

  ccPMInspectDefs::NotTrained
  This **ccPMInspectPattern** is not trained.

  ccPMInspectDefs::BadImage
  The inspection image is unbound.

boundaryDeformation

```
void boundaryDeformation(double value, c_Int32 index);

double boundaryDeformation(c_Int32 index) const;
```

- ```
  void boundaryDeformation(double value, c_Int32 index);
  ```

  Sets the distance in pixels within which feature boundary points can be matched for the inspection region *index*. This parameter indicates how far away a feature boundary point in the run-time region can be from a feature boundary point in the template region and still be considered for a match. The default value is 0.0. The maximum value is 5.0.

Parameters

value The distance in pixels within which feature boundary points can be matched.

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams

value is less than 0.0 or greater than 5.0.
The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

Notes

Calling this function has no effect in this release.

- `double boundaryDeformation(c_Int32 index) const;`

Returns the distance in pixels within which feature boundary points can be matched for the inspection region *index*.

Parameters

index The index for the inspection region.

Throws

ccPMInspectDefs::BadParams

The index for the inspection region is not valid.
Region *index* does not support feature difference mode.

Notes

Calling this function has no effect in this release.

ccPMInspectRegion

```
#include <ch_cvl/pminspct.h>

class ccPMInspectRegion;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encodes all the inspection modes and functions assigned to an inspection region and records all the relative statistical training data. This class cannot be directly instantiated.

Constructors/Destructors

This class cannot be directly instantiated.

A **ccPMInspectRegion** object is created when **ccPMInspectPattern::addInspectRegion()** is called. The call to **ccPMInspectPattern::inspectRegion(c_Int32 Index)** returns the reference to the **ccPMInspectRegion** object corresponding to the region *Index*.

Enumerations

InterpolationEx `enum InterpolationEx`

This enumeration defines the interpolation modes supported for blank scene and intensity difference inspection. This interpolation contains the same values as the **ccAffineSamplingParams::Interpolation** with the addtion of the *eBiliearAccurateFast* value.

To specify the *eBilinearAccurateFast* interpolation mode, you must use the **ccPMInspectRegion::interpolationEx()** function instead of **ccPMInspectRegion::interpolation(),** which only accepts the values defined in **ccAffineSamplingParams::Interpolation**.

Note The second enumeration and function are required because while PatInspect supports the *eBilinearAccurateFast* mode, the Affine Sampling tool, with which the **ccAffineSamplingParams::Interpolation** is shared, does not.

Value	Meaning
eNone = 1	Use nearest-neighbor interpolation.
eBilinear = 2 (Also called *eBilinearApprox*)	Use bilinear interpolation of four nearest pixels. This interpolation mode approximates true bilinear interpolation.
eHighPrecision = 3	Use high-precision interpolation to compute the interpolated value.
eBilinearAccurate = 4	Performs an accurate bilinear interpolation of the four nearest pixels. This method is the most accurate.
eBilinearAccurateFast = 5	Performs an accurate bilinear interpolation of the four nearest pixels. This method produces the same result as *eBilinearAccurate*, but is faster.
kDefaultInterpolation	The default mode, *eHighPrecision*

Public Member Functions

inspectRegion

```
ccPelRect inspectRegion() const;
void inspectRegion(ccAffineRectangle& rect);
```

- `ccPelRect inspectRegion() const;`

  Returns this region. This region is the region referenced by **ccPMInspectPattern::inspectRegion(c_Int32 Index)**.

  If this **ccPMInspectRegion** was constructed using an affine rectangle, this function returns the pixel-aligned bounding box that encloses the region.

- `void inspectRegion(ccAffineRectangle& rect);`

  Places this region into the supplied **ccAffineRectangle** this region.

  If this **ccPMInspectRegion** was constructed using a **ccPelRect**, this function returns the equivalent **ccAffineRectangle**.

  Parameters

  rect The **ccAffineRectangle** into which to place the region.

inspectModes

```
c_UInt32 inspectModes() const;
```

Returns the inspection mode of this inspection region. The value returned is the bitwise OR of enumeration entries from *ccPMInspectDefs::InspectMode* and *ccPMInspectDefs::InspectModeModifier* chosen for this region.

enable

```
bool enable()const;

void enable(bool newState);
```

- `bool enable() const;`

  Returns *true* if this region is enabled. Returns *false* if this region is disabled.

- `void enable(bool newState);`

  Sets the enable state of this region to *newState*.

  Parameters

  newState If *true*, this region is enabled, if *false* this region is disabled.

maskImage

```
const ccPelBuffer_const<c_UInt8>& maskImage() const;

void maskImage(const ccPelBuffer_const<c_UInt8>&
  maskImage);
```

- `const ccPelBuffer_const<c_UInt8>& maskImage() const;`

  Returns the mask image for this region.

  Notes

  If no mask is specified, the function returns an unbound pelbuffer.

- `void maskImage(const ccPelBuffer_const<c_UInt8>& maskImage);`

  Sets the mask image for this inspection region.

  Parameters

  maskImage The mask for this inspection region

  Throws

  ccPMInspectDefs::BadImage

  maskImage is unbound or its dimensions and offset do not match those of this region.

  Notes

  The grey-levels of the pixels in the mask encodes 'care' and 'don't care' regions according to the following table (for more information on masking, see *Patmax* in *Vision Tools Guide)*.

Pixel value in mask	Type of region
0 - 127	'don't care' regions
128 - 191	Reserved for future use
192 - 255	'care' regions

hasMaskImage

```
bool hasMaskImage() const;
```

Returns *true* if this region has a valid mask image.

interpolation

```
ccAffineSamplingParams::Interpolation interpolation()
 const;

void interpolation(
 ccAffineSamplingParams::Interpolation value);
```

- `ccAffineSamplingParams::Interpolation interpolation() const;`

  Returns the interpolation method for affine transformation applied to this region.

  Throws

  ccPMInspectDefs::BadParams

  An interpolation mode of *ccPMInspectRegion::eBilinearAccurateFast* was previously set through a call to **ccPMInspectRegion::interpolationEx()**.

- `void interpolation(`
  `ccAffineSamplingParams::Interpolation value);`

  Sets the interpolation method for affine transformation to be applied to this region. The default interpolation method is *ccAffineSamplingParams::eHighPrecision.*

  Parameters

  value The interpolation method to be applied to this region.

  Notes

  To specify the *ccPMInspectRegion::eBilinearAccurateFast* interpolation mode, you must call **ccPMInspectRegion::interpolationEx()** instead of this function.

interpolationEx

```
InterpolationEx interpolationEx() const;

void interpolationEx(InterpolationEx value);
```

- `InterpolationEx interpolationEx() const;`

  Returns the extended interpolation method for affine transformation applied to this region. The returned value is a member of **ccPMInspectRegion::InterpolationEx**.

- `void interpolationEx(InterpolationEx value);`

  Sets the extended interpolation method for affine transformation to be applied to this region. The default extended interpolation method is *ccPMInspectRegion::eHighPrecision.*

  Parameters

  value The interpolation method to be applied to this region.

  Notes

  This second enumeration and function are required because while PatInspect supports the *eBilinearAccurateFast* mode, the Affine Sampling tool, with which the **ccAffineSamplingParams::Interpolation** is shared, does not.

inspectGrainLimit

```
double inspectGrainLimit(void) const;
void inspectGrainLimit(double limit);
```

- `double inspectGrainLimit(void) const;`

  Returns the granularity limit to use for feature difference inspection mode. The granularity limit is the size, in pixels, of the radius of interest for feature detection. Values of less than 1.0 are used to handle under-resolved features (using software zoom).

- `void inspectGrainLimit(double limit);`

  Sets the granularity limit to use for feature difference inspection mode. The granularity limit is the size, in pixels, of the radius of interest for feature detection. Specify a value of less than 1.0 if the image contains under-resolved features.

  Parameters

  limit The granularity limit in pixels. *limit* must be greater than 0.0.

  Throws

  ccPMInspectDefs::BadParams

  limit is less than or equal to 0.0; this region is not configured for feature difference inspection; *limit* is less than 1.0 and software zoom is not enabled (either for training or run time); or *limit* is greater than 1.0 and software zoom is enabled (either for training or run time).

  Notes

  The default granularity limit is 1.0. To specify a non-default value, you must call this function before you call **ccPMInspectPattern::statisticTrain()** for the first time.

name

```
ccCvlString name() const;
void name(const ccCvlString& idStr);
```

- `ccCvlString name() const;`

  Returns the string identifier for this region.

- `void name(const ccCvlString& idStr);`

  Sets the string identifier for this region. The default value is the null string.

Parameters

idStr The string identifier for this region

displayFeatures

```
void displayFeatures(ccGraphicList& displayList,
 const ccColor& c = ccColor::greenColor())
 const;
```

Draws the trained boundary inspection pattern of this region in the graphic list *displayList* using color *c*.

Parameters

displayList The graphic list to be used for drawing

c The color to be used for drawing

Throws

ccPMInspectDefs::NotTrained
The pattern has not been trained for feature difference inspection mode.

ccPMInspectRegion

ccPMInspectResult

```
#include <ch_cvl/pminspct.h>

class ccPMInspectResult : public cc_PMResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single PatInspect result. The member functions of the class return the following set of results:

- Intensity difference mode:
  - Correlation coefficient between template and run-time inspection region
  - Thresholded difference region
  - Inspection region found in the run-time inspection image
- Boundary difference mode:
  - List of matched, missing and extra features between template and run-time inspection region
- Blank scene inspection mode:
  - List of extra defect features

For more information on the set of results returned by PatInspect see *PatInspect* chapter in *Vision Tools Guide*.

The member functions of the class can return the inspection results only if the **ccPMInspectPattern** that was used to generate it has not been deleted, retrained or run on another image.

Note See the **cc_PMResult** class for additional functions common to all PatMax result classes. The class is not archiveable because re-loading a PatInspect result does not guarantee that the **ccPMInspectPattern** associated with it has also been loaded.

Constructors/Destructors

ccPMInspectResult

```
ccPMInspectResult();
```

Constructs an empty **ccPMInspectResult** object.

Public Member Functions

pose

```
const cc2Xform& pose(c_Int32 regionIndex) const;
```

Returns a **cc2Xform** describing the specified inspection region. This function only returns a distinct **cc2Xform** if you specified *ccPMInspectDefs::eFineAlign* when you created the **ccPMInspectPattern** used to produce this result.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.

ccPMInspectDefs::PatternNoLongerValid
The pattern used to produce this result has been changed or deleted.

matchImage

```
ccGrey8PelBuf matchImage(c_Int32 regionIndex)
  const;
```

Returns the part of the inspection region *regionIndex* that matches the corresponding template region, transformed to the training image coordinate system.

Parameters

regionIndex The inspection region.

Throws

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.

ccPMInspectDefs::BadImage
The found matching region is not fully in the field of view.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has been changed, deleted or run on another image.

correlationScore

```
double correlationScore(c_Int32 regionIndex,
  ccPMInspectDefs::NormalizationMethod method =
  ccPMInspectDefs::eIdentity) const;
```

The function computes the correlation coefficient between the normalized inspection region *regionIndex* and the corresponding template region. The region is normalized by using the normalization method specified by *method*. The value returned by the function is in the range 0.0 through 1.0.

Parameters

image The index for the inspection region

method The normalization method applied to the run-time region. *method* must be one of the following values:

ccPMInspectDefs::eIdentity
ccPMInspectDefs::eHistogramEqualize
ccPMInspectDefs::eMeanAndStdDev
ccPMInspectDefs::eMatchTails

Throws

ccPMInspectDefs::NotTrained
The region was not trained for intensity difference.

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.

ccPMInspectDefs::BadImage
The found matching region is not fully in the field of view.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has been changed, deleted or run on another image.

diffImage

```
ccGrey8PelBuf diffImage(c_Int32 regionIndex,
  ccPMInspectDefs::NormalizationMethod method =
                                ccPMInspectDefs::eIdentity,
  ccPMInspectDefs::DiffMode diffMode =
                                        ccPMInspectDefs::eAbs,
  bool returnRawDiffImg = false) const;
```

This function calculates a difference image between the model and the result. The difference image includes information about raw intensity differences. The output of this function can then be processed using vision tools such as blob and others. *diffMode* specifies the method for computing the difference:

If *diffMode* = *ccPMInspectDefs::eAbs*

D(x, y) = max(abs(M(x, y) - Templ(x, y)) - Thresh(x, y), 0)

If *diffMode* = *ccPMInspectDefs::eClampNeg*

D(x, y) = max(M(x, y) - Templ(x, y) - Thresh(x, y), 0)

If *diffMode* = *ccPMInspectDefs::eClampPos*

D(x, y) = max(Templ(x, y) - M(x, y) - Thresh(x, y), 0)

Where *M* is **matchImage()** normalized using *method*, *Templ* is the template image and *Thresh* is the threshold image.

Parameters

regionIndex The index for the inspection region.

method The normalization method to apply to the run-time region. *method* must be one of the following values:

ccPMInspectDefs::eIdentity
ccPMInspectDefs::eHistogramEqualize
ccPMInspectDefs::eMeanAndStdDev
ccPMInspectDefs::eMatchTails

diffMode The pixel sign handling mode used in the difference operation. Must be one of the following values:

ccPMInspectDefs::eAbs
ccPMInspectDefs::eClampNeg
ccPMInspectDefs::eClampPos

returnRawDiffImg
When set true, this function returns an intermediate image that can be useful for debugging.

Throws

ccPMInspectDefs::NotTrained
The region was not trained for intensity difference.

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.

ccPMInspectDefs::BadImage
The found matching region is not fully in the field of view.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has been changed, deleted or run on another image.

getBoundaryDiff

```
void getBoundaryDiff(
  ccPMInspectBoundaryData& data,
  c_Int32 regionIndex, bool coarseEval = false) const;
```

Computes the three lists of boundary feature differences (matched, missing, and extra) detected by PatInspect for the inspection region *index* and stores them in *data*.

Parameters

data A **ccPMInspectBoundaryData** object in which the feature lists are stored.

regionIndex The index for the inspection region.

coarseEval If *true*, some small features may be discarded or incorporated within larger features. If *fals*e, the complete feature lists are generated.

Throws

ccPMInspectDefs::NotTrained
The region was not trained for boundary difference inspection.

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has changed or has been deleted.

getBlankSceneMeasurement

```
getBlankSceneMeasurement(ccPMInspectAbsenceData& data,
  c_Int32 regionIndex)const;
```

Evaluates the blankness of the region *index* and stores the extra boundary features results in *data*.

Parameters

data A **ccPMInspectAbsenceData** object in which the blank scene inspection results are stored.

regionIndex The index for the inspection region.

Throws

ccPMInspectDefs::NotTrained
The region was not trained for blank scene inspection.

ccPMInspectDefs::BadParams
The index for the inspection region is not valid.

ccPMInspectDefs::PatternNoLongerValid
The pattern used to produce the result has changed or has been deleted.

Deprecated Members

The functions listed in this section are deprecated in CVL 5.5.1. They are provided for backward compatibility only and will not cause compilation errors in CVL 5.5.1. It is recommended to update your code to use the newer versions. These functions will be obsoleted after CVL 5.5.1, at which time they will cause compilation errors.

matchImage

```
ccPelBuffer<c_UInt8> matchImage(
  const ccPelBuffer_const<c_UInt8>& image) const;
```

Returns the part of the image that matches the template image, transformed to the training image coordinate system.

Parameters

image The inspection image. This argument is ignored in this release.

Throws

ccPMInspectDefs::BadImage
The found template image is not fully in the field of view.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has changed or has been deleted.

correlationScore

```
double correlationScore(
  const ccPelBuffer_const<c_UInt8>& image,
  ccPMInspectDefs::NormalizationMethod method =
  ccPMInspectDefs::eIdentity) const;
```

The function computes the correlation coefficient between the normalized inspection image and the template image. The image is normalized by using the normalization method specified by *method*. The value returned by the function is in the range 0.0 through 1.0.

Parameters

image The inspection image. This argument is ignored in this release.

method The normalization method applied to the run-time image. *method* must be one of the following values:

ccPMInspectDefs::eIdentity
ccPMInspectDefs::eHistogramEqualize
ccPMInspectDefs::eMeanAndStdDev
ccPMInspectDefs::eMatchTails

Throws

ccPMInspectDefs::NotTrained
The region was not trained for intensity difference.

ccPMInspectDefs::BadImage
The found matching image is not fully in the field of view.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has changed or has been deleted.

diffImage

```
ccPelBuffer<c_UInt8> diffImage(
  const ccPelBuffer_const<c_UInt8>& image,
  ccPMInspectDefs::NormalizationMethod method =
  ccPMInspectDefs::eIdentity) const;
```

This function computes the thresholded difference image between the inspection image and the corresponding template image. Each pixel in the difference image is first set to the absolute difference between the corresponding pixels in the template image and the normalized run-time inspection image. The threshold image is then applied to the result, with all pixels with values less than the value of the corresponding pixel in the thresholded image set to 0.

Parameters

image The inspection image. This argument is ignored in this release.

method The normalization method to apply to the run-time image. *method* must be one of the following values:

ccPMInspectDefs::eIdentity
ccPMInspectDefs::eHistogramEqualize
ccPMInspectDefs::eMeanAndStdDev
ccPMInspectDefs::eMatchTails

Throws

ccPMInspectDefs::NotTrained
Intensity differences were not trained for this run.

ccPMInspectDefs::BadImage
The found matching image is not fully in the field of view.

ccPMInspectDefs::PatternNoLongerValid
The **ccPMInspectPattern** used to produce the result has changed or has been deleted.

getSimpleBoundaryDiff

```
void getSimpleBoundaryDiff(
  ccPMInspectBoundaryData& lists,
  bool coarseEval = false);
```

Computes the three lists of boundary feature differences (matched, missing, and extra) detected by PatInspect and stores them in *lists*.

Parameters

lists A **ccPMInspectBoundaryData** into which the feature lists are stored.

coarseEval If *true*, some small features may be discarded or incorporated within larger features. If *false*, the complete feature lists are generated.

Throws

ccPMInspectDefs::NotTrained
Feature differences were not trained for this run.

ccPMInspectResultSet

```
#include <ch_cvl/pminspct.h>

class ccPMInspectResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that stores all the results of PatInspect for each inspection region.

Constructors/Destructors

ccPMInspectResultSet

```
ccPMInspectResultSet();
```

Constructs an empty **ccPMInspectResultSet** object.

Public Member Functions

results

```
const cmStd vector<ccPMInspectResult>& results() const;
```

Returns a reference to a vector whose elements contain the inspection results for each inspection region.

time

```
double time() const;
```

Returns the execution time required to produce this result set in seconds.

draw

```
void draw(ccGraphicList& graphList, c_UInt32 drawMode,
  const cc2Xform& pose = cc2Xform()) const;
```

Appends result graphics for this **ccPMInspectResultSet** to the supplied **ccGraphicList**.

Parameters

graphList	The graphics list to append the graphics to.
drawMode	The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccPMInspectDefs::eDrawDefectBoundary
ccPMInspectDefs::eDrawMatchingBoundary
ccPMInspectDefs::eDrawExtraBoundary
ccPMInspectDefs::eDrawMissingBoundary

pose The pose, in client coordinates, at which to draw the graphics.

Notes

Missing or defect features are drawn in **ccColor::redColor()**, matching features are drawn in **ccColor::blueColor()**, and extra features are drawn in **ccColor::yellowColor()**.

ccPMInspectRunParams

```
#include <ch_cvl/pminspct.h>

class ccPMInspectRunParams : public cc_PMRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters used to control the location of the alignment region during PatInspect run-time inspection.

Note See the **cc_PMRunParams** class for functions common to all PatMax run-time parameter classes.

Constructors/Destructors

ccPMInspectRunParams

```
ccPMInspectRunParams();
```

Constructs a **ccPMInspectRunParams** object with default values.

ccPMInspectRunParams

ccPMInspectSimpleBoundaryDiffData

```
#include <ch_cvl/pminspct.h>

class ccPMInspectSimpleBoundaryDiffData;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the three lists of features computed by PatInspect.

Note See the **cc_PMRunParams** class for functions common to all PatMax run-time parameter classes.

Constructors/Destructors

ccPMInspectSimpleBoundaryDiffData

```
ccPMInspectSimpleBoundaryDiffData();
```

Constructs an empty instance of this class.

Public Member Functions

match

```
const cmStd vector <ccPMInspectMatchedFeature>& match()
  const;
```

Each element of the returned vector represents a feature from the run-time image that was matched in the template image.

missing

```
const cmStd vector <ccPMInspectUnmatchedFeature>& missing()
  const;
```

Each element of the returned vector represents a feature from the template image that was missing from the run-time image.

extra

```
const cmStd vector <ccPMInspectUnmatchedFeature>& extra()
  const;
```

Each element of the returned vector represents a feature that was present in the run-time image but not in the template image.

ccPMInspectSimpleBoundaryDiffData

minDeformation `double minDeformation();`

Returns the minimum deformation in client coordinate system units across all feature boundary points in all matched features.

maxDeformation `double maxDeformation();`

Returns the maximum deformation in client coordinate system units across all feature boundary points in all matched features.

ccPMInspectStatTrainParams

```
#include <ch_cvl/pminspct.h>

class ccPMInspectStatTrainParams : public cc_PMRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters used to control the location of the alignment region during PatInspect inspection training.

Note See the **cc_PMRunParams** class for functions common to all PatMax run-time parameter classes.

Constructors/Destructors

ccPMInspectStatTrainParams

```
ccPMInspectStatTrainParams();
```

Constructs a **ccPMInspectStatTrainParams** object with default values.

ccPMInspectStatTrainParams

ccPMInspectUnmatchedFeature

```
#include <ch_cvl/bndpnts.h>

template <class P> class ccPMInspectUnmatchedFeature :
  public cc_PMInspectFeature<ccPMInspectBP>;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a feature that is present in the trained PatInspect pattern but not in the run-time image or a feature that is present in the run-time image but not in the trained PatInspect pattern.

Note You should not instantiate this class directly.

Constructors/Destructors

ccPMInspectUnmatchedFeature

```
ccPMInspectUnmatchedFeature();
```

Constructs a **ccPMInspectUnmatchedFeature**.

ccPMInspectUnmatchedFeature

ccPMMatchInfo

```
#include <ch_cvl/pmpbase.h>

class ccPMMatchInfo;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains data about how the pattern features matched the runtime image and/or how the runtime image features matched the pattern.

Do not create an instance of this class.

Constructors/Destructors

ccPMMatchInfo `ccPMMatchInfo();`

Default constructor.

Notes

Do not create an instance of this class.

The default copy constructor, assignment operator, and destructor are used.

Public Member Functions

patternFeatures `const ccFeatureletChainSet& patternFeatures() const;`

Returns the trained features in the pattern, where the feature weights are set to the quality of the match to runtime features.

The pattern features are in the client coordinates of the run-time image.

imageFeatures `const ccFeatureletChainSet& imageFeatures() const;`

Get the features in the runtime image, where the feature weights are set to the quality of the match to a pattern feature.

The image features are in the client coordinates of the run-time image.

displayPattern

```
void displayPattern(ccUITablet& tablet) const;
void displayPattern(ccGraphicList& graphList) const;
```

- `void displayPattern(ccUITablet& tablet) const;`

  Generates a display in the specified **ccUITablet** object showing how the pattern matches the image.

  Parameters

  tablet The tablet where the display is created.

- `void displayPattern(ccGraphicList& graphList) const;`

  Generates a display in the specified **ccGraphicList** object showing how the pattern matches the image.

  Parameters

  graphList The display in graphics list format.

  Notes

  Red features indicate a poor match, yellow ones indicate fair match, and green ones indicate good match, as indicated by the following table.

Feature weight	Quality	Color
[0.00, 0.20)	poor	red
[0.20, 0.67)	fair	yellow
[0.67, 1.00]	good	green

displayImage

```
void displayImage(ccUITablet& tablet) const;
void displayImage(ccGraphicList& graphList) const;
```

- `void displayImage(ccUITablet& tablet) const;`

  Generates a display in the specified **ccUITablet** object showing how the image matches the pattern.

  Parameters

  tablet The tablet where the display is created.

- `void displayImage(ccGraphicList& graphList) const;`

  Generates a display in the specified **ccGraphicList** object showing how the image matches the pattern.

  Parameters

  graphList The display in graphics list format.

  Notes

  Red features indicate a poor match, yellow ones indicate fair match, and green ones indicate good match, as indicated by the following table.

Feature weight	Quality	Color
[0.00, 0.20)	poor	red
[0.20, 0.67)	fair	yellow
[0.67, 1.00]	good	green

ccPMMatchInfo

ccPMMultiModel

```
#include <ch_cvl/pmmm.h>

class ccPMMultiModel;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a PatMax Multi-Model.

Constructors/Destructors

ccPMMultiModel

```
ccPMMultiModel();

~ccPMMultiModel();

ccPMMultiModel(const ccPMMultiModel& rhs);
```

- `ccPMMultiModel();`

  Construct an untrained PatMax Multi-Model with default values.

- `~ccPMMultiModel();`

  Destructs the PatMax Multi-Model.

- `ccPMMultiModel(const ccPMMultiModel& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

Operators

operator=

```
ccPMMultiModel& operator=(const ccPMMultiModel& rhs);
```

Assignment operator.

Parameters

rhs Source of the assignment.

Public Member Functions

addModel

```
c_Int32 addModel(
const ccPMAlignPattern& model,
ccPMMultiModelDefs::GrainLimitPolicy grainLimitPolicy =
ccPMMultiModelDefs::kMustMatch,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);
```

Adds a traditionally-trained PatMax model to the Multi-Model and returns the internally assigned id. This model must not have been trained using Composite Model training. See **addCompositeModel()** for that use case.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kMustMatch*, then the coarse and fine granularity limits of the passed PatMax model and those of the component PatMax models already contained in the **ccPMMultiModel** instance are required to match.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kRetrainAtFinest*, then the granularity limits of the passed PatMax model and those already contained in the **ccPMMultiModel** instance will be examined. The finest coarse granularity limit and the finest fine granularitygrain limit will then be used to either retrain a copy of the passed PatMax model or all the existing PatMax models contained in the Multi-Model or both. Any model or models that are already trained for these granularity limits will not be affected.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kRetrainExistingWithNew* and the passed model's coarse and/or fine granularity limit is/are different from the granularity limits of the component PatMax models already contained in the **ccPMMultiModel** instance, then all these models will be retrained at the passed model's granularity limits.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kRetrainNewWithExisting* and the passed model's coarse and/or fine granularity limit is/are different from the granularity limits of the component PatMax models already contained in the **ccPMMultiModel** instance, then a copy of the passed PatMax model will be retrained with the granularity limits of the models already contained in the **ccPMMultiModel** instance and before being added to the **ccPMMultiModel** instance.

Parameters

model The PatMax model to add.

grainLimitPolicy The granularity limit policy.

diagobj A diagnostic object.

diagFlags Diagnostic flags.

Notes

This function makes a deep copy of the passed **ccPMAlignPattern** object.

Throws

MissingTrainingImage

The passed model was trained with **ccPMAlignPattern::saveImage()** == false.

BadParams

grainLimitPolicy == ccPMMultiModelDefs::kMustMatch and either the model's **coarseGrainLimit()** and/or its **fineGrainLimit()** does not match that of the models already added to (contained in) the Multi-Model.
OR
The passed model was trained using Composite Model training.

addCompositeModel

```
c_Int32 addCompositeModel(
const ccPMCompositeModelManager& compModelMgr,
const ccPMCompositeModelParams& params,
ccPMMultiModelDefs::GrainLimitPolicy grainLimitPolicy =
ccPMMultiModelDefs::kMustMatch,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);
```

Adds a Composite PatMax Model to the Multi-Model and returns the internally assigned id. The Multi-Model will make a copy of the passed **ccPMCompositeModelManager** and **ccPMCompositeModelParams** and use these to train an internal PatMax model.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kMustMatch*, then the coarse and fine granularity limits of the passed Composite Model Manager and those of the component PatMax models already contained in the **ccPMMultiModel** instance are required to match.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kRetrainAtFinest*, then the granularity limits of the passed Composite Model Manager and those already contained in the **ccPMMultiModel** instance will be examined. The finest coarse granularity limit and the finest fine granularity limit will then be used to either retrain a copy of the passed

Composite Model Manager or all the existing PatMax models contained in the Multi-Model or both. Any model or models that are already trained for these granularity limits will not be affected.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kRetrainExistingWithNew*, then if the passed Composite Model Manager's coarse and/or fine granularity limit is/are different from the granularity limits of the component PatMax models already contained in the **ccPMMultiModel** instance, then all these models will be retrained at the passed model's granularity limits.

If *grainLimitPolicy* equals *ccPMMultiModelDefs::kRetrainNewWithExisting*, then if the passed Composite Model Manager's coarse and/or fine granularity limit is/are different from the granularity limits of the component PatMax models already contained in the **ccPMMultiModel** instance, then a copy of the passed Composite Model Manager will be retrained with the granularity limits of the models already contained in the **ccPMMultiModel** instance and before being added to the **ccPMMultiModel** instance.

Parameters

compModelMgr	The Composite Model Manager to be added.
params	The Composite Model parameters.
grainLimitPolicy	The granularity limit policy.
diagobj	A diagnostic object.
diagFlags	Diagnostic flags.

Notes

This function makes deep copies of the passed **ccPMCompositeModelManager** and **ccPMCompositeModelParams** objects.

Throws

BadParams

grainLimitPolicy == *ccPMMultiModelDefs::kMustMatch* and either the Composite Model Manager's coarse or fine granularity does not match that of the models already added to (contained in) the Multi-Model.

MissingTrainingImage

The passed *grainLimitPolicy* != *ccPMMultiModelDefs::kMustMatch* and the passed Composite Model Manager or one of any previously added Composite Model Managers requires retraining and was originally trained with **ccPMCompositeModelManager::saveImages()** == false.

numModels

```
c_Int32 numModels() const;
```

Returns the number of component models in the Multi-Model.

The default is 0.

getModelIds

```
cmStd vector<c_Int32> getModelIds() const;
```

Returns a list of model id's for all models contained in the Multi-Model

The default is an empty vector.

model

```
const ccPMAlignPattern& model(c_Int32 modelId) const;
```

Returns a reference to the model corresponding to the supplied id.

Parameters

modelId The model id you supply.

Throws

BadParams

The passed *modelId* is invalid.

removeModel

```
void removeModel(c_Int32 modelId);
```

Removes a model from the Multi-Model. If the Multi-Model owns the model to be removed, then the model will be deleted from the memory.

Parameters

modelId The model id of the model to be removed.

Throws

BadParams

The passed *modelId* is invalid.

resultStatisticWindowLength

```
c_Int32 resultStatisticWindowLength() const;
void resultStatisticWindowLength(c_Int32 length);
```

- `c_Int32 resultStatisticWindowLength() const;`

Returns the window over which result statistics are collected.

- `void resultStatisticWindowLength(c_Int32 length);`

  Sets the window over which result statistics are collected.

  Result statistics are used to compute the most successful model when the run-time mode is set to *ccPMMultiModelDefs::kSequentialMostSuccessful*.

  The default is 10.

  Parameters
  length The length of the window.

  Notes
  Calling the setter causes any existing result statistics to be flushed.

  Throws
  BadParams
  length <= 0

resetResultStatistics

`void resetResultStatistics();`

Resets the statistics that are used to compute the most successful model when the run-time mode is set to *ccPMMultiModelDefs::kSequentialMostSuccessFul*.

modelIdQueue

`cmStd vector<c_Int32> modelIdQueue() const;`

`void modelIdQueue(cmStd vector<c_Int32> modelIdQueue);`

- `cmStd vector<c_Int32> modelIdQueue() const;`

  Returns the model queue used in all sequential run-time modes.

- `void modelIdQueue(cmStd vector<c_Int32> modelIdQueue);`

  Sets the model queue used in all sequential run-time modes.

  It is intended that you may modify the queuing sequence to create model queuing sequences tailored to their application. The queuing sequence may be modified between calls to **ccPMMultiModel::run()** allowing you to create dynamic queuing schemes.

  The default is an empty vector.

  Parameters
  modelIdQueue The model queue.

Notes

The *modelIdQueue* does not have to contain entries for all the models contained in the Multi-Model.

If runtime parameter **ccPMMultiModelRunParams::mode()** is *ccPMMultiModelDefs::kSequentialMostRecentlySuccessful* or *ccPMMultiModelDefs::kSequentialMostSuccessful*, then the *modelIdQueue* will be updated during a call to run.

If **ccPMMultiModelRunParams::mode()** is *ccPMMultiModelDefs::kSequentialMostRecentlySuccessful*, then the queue will be re-ordered so that the model achieving the highest score moves to the front.

If **ccPMMultiModelRunParams::mode()** is *ccPMMultiModelDefs::kSequentialMostSuccessful*, then the queue will be re-ordered so that the models are ranked in order of success tabulated over the previous **resultStatisticWindowLength()** calls to run.

Throws

BadParams

modelIdQueue is supplied with a model id that does not correspond to a model contained within the Multi-Model, *or modelIdQueue* is supplied with a model id vector that contains duplicate entries for the same model.

---

run

```
void run(
const ccPelBuffer_const<c_UInt8>& inputImage,
const ccPMMultiModelRunParams& params,
ccPMMultiModelResultSet& resultSet,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);

void run(
const ccPelBuffer_const<c_UInt8>& inputImage,
const ccPelBuffer_const<c_UInt8>& mask,
const ccPMMultiModelRunParams& params,
ccPMMultiModelResultSet& resultSet,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);

void run(
const ccPelBuffer_const<c_UInt8>& inputImage,
const ccPMMultiModelRunParams& params,
const cc2XformBasePtrh_const& startPose,
```

```
ccPMMultiModelResultSet& resultSet,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);

void run(
const ccPelBuffer_const<c_UInt8>& inputImage,
const ccPelBuffer_const<c_UInt8>& mask,
const ccPMMultiModelRunParams& params,
const cc2XformBasePtrh_const& startPose,
ccPMMultiModelResultSet& resultSet,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);
```

- ```
  void run(
  const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPMMultiModelRunParams& params,
  ccPMMultiModelResultSet& resultSet,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);
  ```

  Run this Multi-Model on the given image with the given run-time parameters. The supplied *resultSet* will be cleared and then filled with new results, in order of decreasing score.

  If the run-time mode is equal to *ccPMMultiModelDefs::kSequentialMostRecentlySuccessful* or *ccPMMultiModelDefs::kSequentialMostSuccessful*, then the internal queuing sequence of the component models will be updated with the results from this call to run.

  Parameters

inputImage	The input image.
params	The Multi-Model run-time parameters.
resultSet	The result set.
diagobj	A diagnostic object.
diagFlags	Diagnostic flags.

  Notes

  The function is terminated if it runs longer than the timeout specified by the *params* argument. If this occurs, no valid results are produced by the tool.

  Throws

NotTrained	This Multi-Model is empty or any of the component models is not currently trained.
  ```

BadImage
The *inputImage* is unbound.

BadParams
Any of the component models was not trained for the algorithm selected by the *params* argument,
or run-time parameter
ccPMMultiModelRunParams::isModeSequential() == true and run-time parameters
ccPMMultiModelRunParams::confusionThreshold() < **ccPMMultiModelRunParams::acceptThreshold()**

NotImplemented
ccPMMultiModelRunParams::algorithm is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp*.

ccTimeout::Expired
This function executes longer than the timeout you specified.

- ```
  void run(
  const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccPMMultiModelRunParams& params,
  ccPMMultiModelResultSet& resultSet,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);
  ```

  Run this Multi-Model on the given image with the given run-time parameters. The supplied *resultSet* will be cleared and then filled with new results, in order of decreasing score.

  If the run-time mode is equal to *ccPMMultiModelDefs::kSequentialMostRecentlySuccessful* or *ccPMMultiModelDefs::kSequentialMostSuccessful*, then the internal queuing sequence of the component models will be updated with the results from this call to run.

  Parameters

inputImage	The input image.
mask	The mask used to specify care pixels in the input image.
params	The Multi-Model run-time parameters.
resultSet	The result set.
diagobj	A diagnostic object.
diagFlags	Diagnostic flags.
  ```

Notes

In the *mask*, pixels less than 128 indicate "don't care" pixels in the search image. Those greater than or equal to 128 denote "care" pixels in the search image. All feature boundary points that fall within "don't care" pixels are discarded.

The function is terminated if it runs longer than the timeout specified by the *params* argument. If this occurs, no valid results are produced by the tool.

Throws

NotTrained

This Multi-Model is empty or any of the component models is not currently trained.

BadImage

The *inputImage* is unbound,
or the *mask* is unbound or the *mask*'s region is not equal to the *inputImage*'s region.

BadParams

Any of the component models was not trained for the algorithm selected by the *params* argument,
or run-time parameter
ccPMMultiModelRunParams::isModeSequential() == true and run-time parameters
ccPMMultiModelRunParams::confusionThreshold() < **ccPMMultiModelRunParams::acceptThreshold()**

NotImplemented

ccPMMultiModelRunParams::algorithm is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp*.

ccTimeout::Expired

This function executes longer than the timeout you specified.

-
```
void run(
const ccPelBuffer_const<c_UInt8>& inputImage,
const ccPMMultiModelRunParams& params,
const cc2XformBasePtrh_const& startPose,
```

```
ccPMMultiModelResultSet& resultSet,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);
```

Using the supplied starting pose as a coarse pose of the pattern in the *inputImage*, this method returns a further refined search results based on the given run-time parameters

The supplied *resultSet* will be cleared and then filled with the single new result.

If the run-time mode is equal to *ccPMMultiModelDefs::kSequentialMostRecentlySuccessful* or *ccPMMultiModelDefs::kSequentialMostSuccessful*, then the internal queuing sequence of the component models will be updated with the results from this call to run.

Parameters

inputImage	The input image.
params	The Multi-Model run-time parameters.
startPose	The start pose.
resultSet	The result set.
diagobj	A diagnostic object.
diagFlags	Diagnostic flags.

Notes

startPose is a client-to-client transform representing a coarse initial pose which is subsequently refined.

The function is terminated if it runs longer than the timeout specified by the *params* argument. If this occurs, no valid results are produced by the tool.

Throws

NotTrained

This Multi-Model is empty or any of the component models is not currently trained.

BadImage

The *inputImage* is unbound.

BadParams

This pattern was not trained for the algorithm selected by the *params* argument,
or the run-time parameter
ccPMMultiModelRunParams::isModeSequential() == true and run-time parameters

ccPMMultiModelRunParams::confusionThreshold() < **ccPMMultiModelRunParams::acceptThreshold()**, *or startPose* == NULL

NotImplemented
ccPMMultiModelRunParams::algorithm is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp*, *or ccPMMultiModelRunParams::algorithm* is *cc_PMDefs::kPatquick* and *startPose* != NULL, *or* ! **startPose.isLinear()**

ccTimeout::Expired
This function executes longer than the timeout you specified.

- ```
  void run(
  const ccPelBuffer_const<c_UInt8>& inputImage,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccPMMultiModelRunParams& params,
  const cc2XformBasePtrh_const& startPose,
  ccPMMultiModelResultSet& resultSet,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);
  ```

  Using the supplied starting pose as a coarse pose of the pattern in the *inputImage*, this method returns a further refined search results based on the given run-time parameters.

  The supplied *resultSet* will be cleared and then filled with the single new result.

  If the run-time mode is equal to *ccPMMultiModelDefs::kSequentialMostRecentlySuccessful* or *ccPMMultiModelDefs::kSequentialMostSuccessful*, then the internal queuing sequence of the component models will be updated with the results from this call to run.

  Parameters

inputImage	The input image.
mask	The mask used to specify care pixels in the input image.
params	The Multi-Model run-time parameters.
startPose	The start pose.
resultSet	The result set.
diagobj	A diagnostic object.
diagFlags	Diagnostic flags.
  ```

Notes

startPose is a client-to-client transform representing a coarse initial pose which is subsequently refined.

In the mask, pixels less than 128 indicate "don't care" pixels in the search image. Those greater than or equal to 128 denote "care" pixels in the search image. All feature boundary points that fall within "don't care" pixels are discarded.

The function is terminated if it runs longer than the timeout specified by the *params* argument. If this occurs, no valid results are produced by the tool.

Throws

NotTrained

This Multi-Model is empty or any of the component models is not currently trained.

BadImage

The *inputImage* is unbound,
or the *mask* is unbound or the *mask*'s region is not equal to the *inputImage*'s region.

BadParams

This pattern was not trained for the algorithm selected by the params argument,
or the run-time parameter
ccPMMultiModelRunParams::isModeSequential() == true and run-time parameters
ccPMMultiModelRunParams::confusionThreshold() < **ccPMMultiModelRunParams::acceptThreshold()**,
or startPose == NULL

NotImplemented

ccPMMultiModelRunParams::algorithm is *cc_PMDefs::kPatflex* or *cc_PMDefs::kPatpersp*,
or ccPMMultiModelRunParams::algorithm is *cc_PMDefs::kPatquick* and *startPose* != NULL,
or ! **startPose.isLinear()**

ccTimeout::Expired

This function executes longer than the timeout you specified.

ccPMMultiModel

ccPMMultiModelDefs

```
#include <ch_cvl/pmmm.h>

class ccPMMultiModelDefs;
```

A name space that holds enumerations used with the Multi-Model PatMax tool.

Enumerations

RuntimeMode `enum RuntimeMode;`

This enumeration defines the run-time mode.

Value	Meaning
kSequential = 1	Models are run in a predefined order (queue) until enough results are found with scores greater than or equal to a user-defined confusion threshold to satisfy the PatMax **ccPMMultiModelRunParams::numToFind()** run-time parameter. You define the confusion threshold with **ccPMMultiModelRunParams::confusionThreshold()**. A separate run-time parameter, **ccPMMultiModelRunParams::useXYOverlapBetweenModels()** determines whether lower scoring overlapping results are discarded and are therefore not counted towards the total applied to the **ccPMMultiModelRunParams::numToFind()** threshold. A further run-time parameter, **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** can be used to allow results from only one PatMax model (within the PatMax Multi-Model) to be counted towards the **ccPMMultiModelRunParams::numToFind()** threshold. (This is intended to address the use case where the application knows that it wants to find/count multiple parts and the parts may be of various types, for example, type A, type B, type C, and so on. In addition, the application has apriori knowledge that at any one time the field of view will contain parts of only one type. The sequence of the type, however, cannot be predicted with certainty.) The order of model execution (queuing sequence) is settable through the external API and this may be done after each run-time call, if so desired. If no models score greater than or equal to the **ccPMMultiModelRunParams::confusionThreshold()**, then the single model results from the highest scoring model are returned.
kSequentialMostRecentlySuccessful = 2	A variant on sequential mode, where after each call to run the successful model is promoted to first in the queuing sequence.

Value	Meaning
kSequentialMostSuccessful = 3	A variant on sequential mode, where after each call to PatMax Multi-Model run-time, a histogram tabulating the successful model id's of the previous N calls run-time is updated (N is an integer you define that provides the temporal window over which model "success" is evaluated). The queuing sequence is then set to reflect the relative values in this histogram.
kExhaustive = 4	All models in a Multi-Model are executed simultaneously (using as many processor cores as are enabled by the work manager – see *ch_cvl/workmgr.h*). The reported result set includes single model results either from all models (if **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** is false) or just a single model (if **ccPMMultiModelRunParams::reportResultsFromOneModelOnly()** is true). The results are sorted in order of highest score first. The component model used to generate each individual result is identifiable through the API.

GrainLimitPolicy

```
enum GrainLimitPolicy;
```

This enumeration defines the granularity limit policy.

Value	Meaning
kMustMatch = 0	The coarse and fine granularity limits of the passed PatMax model (or Composite Model Manager) and those of the component PatMax models already contained in the **ccPMMultiModel** instance are required to match.
kRetrainAtFinest = 1	The granularity limits of the passed PatMax model (or Composite Model Manager) and those already contained in the **ccPMMultiModel** instance will be examined. The finest coarse granularity limit and the finest fine granularitygrain limit will then be used to either retrain a copy of the passed PatMax model (or Composite Model Manager) or all the existing PatMax models contained in the Multi-Model or both. Any model or models that are already trained for these granularity limits will not be affected.
kRetrainExistingWithNew = 2	If the passed model's (or Composite Model Manager's) coarse and/or fine granularity limit is/are different from the granularity limits of the component PatMax models already contained in the **ccPMMultiModel** instance, then all these models will be retrained at the passed model's granularity limits.
kRetrainNewWithExisting = 3	If the passed model's (or Composite Model Manager's) coarse and/or fine granularity limit is/are different from the granularity limits of the component PatMax models already contained in the **ccPMMultiModel** instance, then a copy of the passed PatMax model (or Composite Model Manager) will be retrained with the granularity limits of the models already contained in the **ccPMMultiModel** instance and before being added to the **ccPMMultiModel** instance.

ccPMMultiModelDefs

ccPMMultiModelResultSet

```
#include <ch_cvl/pmmm.h>

class ccPMMultiModelResultSet : public ccPMAlignResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a PatMax Multi-Model result set.

Constructors/Destructors

ccPMMultiModelResultSet

```
ccPMMultiModelResultSet();
```

Constructs an empty result set.

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

draw

```
void draw(
ccGraphicList& graphList,
c_UInt32 drawMode = ccPMAlignDefs::eDrawStandard,
bool drawOnlyAccepted = false) const;
```

Appends graphics for all results to *graphList*. *drawMode* is used to select which graphics are drawn (see the comment in **ccPMAlignResult::draw()**).

The *drawOnlyAccepted* argument, if true, causes this function to append graphics of only the accepted results. If it is false, graphics of all results are appended.

Parameters

graphList The graphics list.

drawMode The draw mode.

drawOnlyAccepted
Whether to append graphics of only the accepted results.

trainMethod `ccPMAlignDefs::TrainMethod trainMethod(c_Int32 modelId);`

Returns the training method for the supplied model ID.

Parameters

modelId The model ID.

Friends

ccPMMultiModel

```
friend class ccPMMultiModel;
```

ccPMAlignResult

```
friend class ccPMAlignResult;
```

ccPMMultiModelRunParams

```
#include <ch_cvl/pmmm.h>

class ccPMMultiModelRunParams : public ccPMAlignRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains PatMax Multi-Model run-time parameters.

Constructors/Destructors

ccPMMultiModelRunParams

```
ccPMMultiModelRunParams();
```

Constructs a **ccPMMultiModelRunParams** object using the default values.

Notes

The compiler-generated copy constructor, assignment operator, and destructor are used.

Public Member Functions

mode

```
ccPMMultiModelDefs::RuntimeMode mode() const;

void mode(ccPMMultiModelDefs::RuntimeMode mode);
```

- ```
  ccPMMultiModelDefs::RuntimeMode mode() const;
  ```

  Returns the run-time mode.

- ```
  void mode(ccPMMultiModelDefs::RuntimeMode mode);
  ```

  Sets the run-time mode.

  The default is *ccPMMultiModelDefs::kSequential*

  Parameters

mode	The run-time mode to be set.

Throws

BadParams

The setter is given a value that is not a member of **ccPMMultiModelDefs::RuntimeMode**.

isModeSequential

```
bool isModeSequential() const;
```

Gets whether the run-time mode is sequential.

The default is true.

isModeExhaustive

```
bool isModeExhaustive() const;
```

Gets whether the run-time mode is exhaustive.

The default is false.

confusionThreshold

```
double confusionThreshold() const;
void confusionThreshold(double threshold);
```

- `double confusionThreshold() const;`

  Returns the confusion threshold.

- `void confusionThreshold(double threshold);`

  Sets the confusion threshold.

  If **isModeSequential()** is true, then the confusion threshold is used to determine the stopping condition of the sequential search, that is, if the total number of results found scoring greater than or equal to **confusionThreshold()** is greater than or equal to **numToFind()**, then no further PatMax models within a **ccPMMultiModel** instance will be run.

  The default is 0.5 (which is identical to the default **acceptThreshold()**).

  Parameters

  threshold The confusion threshold to be set.

Notes

If **isModeSequential()** equals true and **confusionThreshold()** is less than **acceptThreshold()**, then the **ccPMMultiModel** run functions will throw *BadParams*

The confusion threshold is not used if **isModeSequential()** == false.

Throws

BadParams

The setter is given a value < 0 or > 1

useXYOverlapBetweenModels

```
bool useXYOverlapBetweenModels() const;
void useXYOverlapBetweenModels(bool enabled);
```

- `bool useXYOverlapBetweenModels() const;`

  Returns whether the **xyOverlap()** threshold will be applied to results generated by different models run from a **ccPMMultiModel** instance.

- `void useXYOverlapBetweenModels(bool enabled);`

  Sets whether the **xyOverlap()** threshold will be applied to results generated by different models run from a **ccPMMultiModel** instance.

  If **useXYOverlapBetweenModels()** equals true and two results from different models overlap by greater than **xyOverlap()**, then only the higher scoring of these results will be returned in the result set. Also, if **isModeSequential()** equals true, **useXYOverlapBetweenModels()** equals true, and two results from different models overlap by greater than **xyOverlap()**, then only the higher scoring of these results will count towards reaching the **numToFind()** stopping condition of the sequential search.

  The default is true.

  Parameters

  enabled — Whether the **xyOverlap()** threshold will be applied to results generated by different models run from a **ccPMMultiModel** instance.

reportResultsFromOneModelOnly

```
bool reportResultsFromOneModelOnly() const;

void reportResultsFromOneModelOnly(bool enabled);
```

- `bool reportResultsFromOneModelOnly() const;`

  Returns whether results are to be reported from one component PatMax model contained within a **ccPMMultiModel** instance.

- `void reportResultsFromOneModelOnly(bool enabled);`

  Sets whether results are to be reported from one component PatMax model contained within a **ccPMMultiModel** instance.

  If **reportResultsFromOneModelOnly()** equals true and **isModeSequential()** equals true, then only results from one model will be reported and only results from one model will count towards reaching the **numToFind()** stopping condition of the sequential search.

  If **reportResultsFromOneModelOnly()** equals true and no component PatMax models generate greater than or equal to **numToFind()** results with scores greater than or equal to **acceptThreshold()**, then the results from the single model with the most results scoring greater than or equal to **acceptThreshold()** will be returned. In the event of a tie, results from the model with the highest single score will be returned.

  The default is false.

  Parameters

enabled	Whether results are to be reported from one component PatMax model contained within a **ccPMMultiModel** instance.

ccPMPerspectiveResult

```
#include <ch_cvl/pmpbase.h>

class ccPMPerspectiveResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single PatPersp result. You should not instantiate objects of this class in your programs. **ccPMPerspectiveResult** objects are created for you as needed.

To run PatPersp, prepare PatMax using the *kPatpersp* algorithm. You run PatPersp by calling **ccPMAlignPattern::run()** as you would for other PatMax algorithms.

PatMax runs using the PatPersp algorithm and produces a PatPersp result along with its normal results. You access the PatPersp result with the getter **cc_PMResult::perspectiveResult()**.

The PatPersp result contains the **ccPMPerspectiveResult::perspectiveXformRunImageFromTrainImage()** deformation transform that you can apply to the run-time image to transform it into an image that matches the trained model. Cognex provides the global function **cfSampledImageWarp()** you can use for this transformation.

Constructors/Destructors

The default constructor, destructor, copy constructor, and assignment operator are used. Do not instantiate this class in your programs. Objects are created for you as needed.

Public Member Functions

perspectivePose

```
cc2XformPerspectivePtrh_const perspectivePose() const;
```

Returns a perspective transform that maps the pattern of the run-time image to the client coordinates of the run-time image. This is a pattern-coordinates-to-client-coordinates transform.

Notes

If not the PatPersp algorithm is run or PatPersp fails to find any result, this will be a null pointer-handle.

perspectiveXformRunImageFromTrainImage

```
cc2XformPerspectivePtrh_const
perspectiveXformRunImageFromTrainImage() const;
```

Returns a perspective transform that maps the training image to the image coordinates of the run-time image. This is an image-coordinates-to-image-coordinates transform.

Notes

If not the PatPersp algorithm is run or PatPersp fails to find any result, this will be a null pointer-handle.

ccPNG

```
#include <ch_cvl/png.h>

class ccPNG;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class defines the API for loading and storing various types of images in CVL in Portable Network Graphics (PNG) format, including:

a. 8-bit images

b. 16-bit images

c. 3-channel 8-bit color images (encoded as 24-bit packed)

d. 4-channel 8-bit color images (encoded as 32-bit packed)

e. 3-plane 8-bit images

PNG is a lossless image compression format.

The API provides interface to the PNG image file's Signficant Bits (the number of significant bits encoded in a PNG image file) - via **ccPNG::numSignificantBits()**, a getter function.

Notes

The user can only set the *numSignificantBits* value for 16-bit images (that is, **numSignificantBits()** is automatically set to 8 for 8-bit images, 3-plane 8-bit images, 3-channel 8-bit color images, and 4-chanel 8-bit color images).

Each image's image-coordinate offset is stored into/loaded from the PNG file. In other words, the PNG file contains the image coordinate offset.

Each image's client-coordinate transform is not stored into, nor loaded from, the PNG file. In other words, the PNG file does not contain the client coordinate transform. Consequently, images loaded from PNG data always have identity client coordinates.

The **pelBuffer()** getter functions return deep copies of the pel buffers.

Enumerations

ImageType

```
enum ImageType;
```

These values specify the image types.

Value	Meaning
eImage8Bit	8 bit image type.
eImage16Bit	16 bit image type.
eImageColor	Color image type.

Constructors/Destructors

ccPNG

```
ccPNG();
```

Constructs an uninitialized **ccPNG** object with no image pixel data and *numSignificantBits* == 0.

Notes

Compiler-generated copy constructor and assignment operator are used.

Public Member Functions

init

```
void init(const ccCvlString& filename);

void init(cmStd istream&);

void init(const ccPelBuffer_const<c_UInt8>& pelBuf);

void init(const ccPelBuffer_const<c_UInt16>& pelBuf,
  c_Int32 numSignificantBits = 16);

void init(const ccPelBuffer_const<ccPackedRGB32Pel>&
  pelBuf);

void init(const cc3PlanePelBuffer_const& pelBuf);
```

- `void init(const ccCvlString& filename);`

Reads a PNG from the given filename and initializes this object with pixel data.

Parameters

filename The name of the file.

Notes

Sets the *numSignificantBits* value from the file.

Sets the *imageType* according to the file content.

Throws

ccPNG::ReadError
The given file cannot be opened or read.

ccPNG::BadFormat
The given file is not in the proper PNG format.

- `void init(cmStd istream&);`

Reads a PNG from the given istream and initializes this object with pixel data. This operation moves the stream position to the end of this PNG data.

Parameters

istream The given istream.

Notes

Sets the *numSignificantBits* value from the stream.

Sets the *imageType* according to the PNG content of the stream.

Throws

ccPNG::ReadError
The given istream is not in the proper PNG format.

- `void init(const ccPelBuffer_const<c_UInt8>& pelBuf);`

Given a pelBuffer, initializes this object with image data from the pelBuffer.

Parameters

pelBuf The given pelBuffer.

Notes

Sets the *numSignificantBits* to 8.

Sets the *imageType* to *eImage8Bit*.

The image's image-coordinate offset is stored.

The image's client-coordinate transform is not stored.

Throws

ccPel::UnboundWindow
pelBuf is not bound.

- `void init(const ccPelBuffer_const<c_UInt16>& pelBuf, c_Int32 numSignificantBits = 16);`

  Given a pelBuffer, initializes this object with image data from the pelBuffer, and stores the provided *numSignificantBits* value.

  Parameters

  pelBuf The given pelBuffer.

  numSignificantBits
  The number of significant bits.

  Notes

  Sets the *imageType* to *eImage16Bit.*

  The image's image-coordinate offset is stored.

  The image's client-coordinate transform is not stored.

  Throws

  ccPNG::BadParams
  numSignificantBits < 8 OR *numSignificantBits* > 16

  ccPel::UnboundWindow
  pelBuf is not bound.

  ccPNG::BadParams
  Any pel in pelBuf has greylevel greater than (or equal to) 1 << *numSignificantBits*.

- `void init(const ccPelBuffer_const<ccPackedRGB32Pel>& pelBuf);`

  Given a pelBuffer, initializes this object with image data from the pelBuffer.

  Parameters

  pelBuf The given pelBuffer.

  Notes

  Sets the *numSignificantBits* to 8.

  Sets the *imageType* to *eImageColor*.

  The image's image-coordinate offset is stored.

  The image's client-coordinate transform is not stored.

  Throws

ccPel::UnboundWindow
pelBuf is not bound.

- `void init(const cc3PlanePelBuffer_const& pelBuf);`

Given a pelBuffer, initializes this object with image data from the pelBuffer.

Parameters
pelBuf The given pelBuffer.

Notes
Sets the *numSignificantBits* to 8.

Sets the *imageType* to *eImageColor*.

The image's image-coordinate offset is stored.

The image's client-coordinate transform is not stored.

Throws
ccPel::UnboundWindow
pelBuf is not bound

imageType `ImageType imageType() const;`

Returns the image type associated with this **ccPNG** object.

Throws
ccPNG::NoPngData
This **ccPNG** was default constructed.

---

write `void write(const ccCvlString& filename) const;`

`void write(cmStd ostream&) const;`

- `void write(const ccCvlString& filename) const;`

Writes this **ccPNG** object out to the given file in PNG format.

Parameters
filename The name of the file.

Throws
ccPNG::NoPngData
This **ccPNG** was default constructed.

- `void write(cmStd ostream&) const;`

Writes this **ccPNG** object out to the given stream in PNG format.

Parameters

ostream The given ostream.

Throws

ccPNG::WriteError
The given stream cannot be written.

ccPNG::NoPngData
This **ccPNG** was default constructed.

---

pelBuffer

```
ccPelBuffer<c_UInt8> pelBuffer() const;

void pelBuffer(ccPelBuffer<c_UInt16>& pelBuf) const;

void pelBuffer(ccPelBuffer<ccPackedRGB32Pel>& pelBuf)
 const;

void pelBuffer(cc3PlanePelBuffer& pelBuf) const;
```

- `ccPelBuffer<c_UInt8> pelBuffer() const;`

Returns a pelBuffer representation of this **ccPNG** object.

Notes

A 16-bit pel buffer (or PNG file) is scaled down to 8 bits by rightshifting by **numSignficantBits()**-8 and then clamped to 255.

A color pel buffer (or PNG file) is converted to greyscale. The algorithm used for converting color PNGs to 8-bit images is as follows:
pixel value = (Red + Green + Blue) / 3

The image's image-coordinate offset is read from the data.

The image's client-coordinate transform is identity.

Throws

ccPNG::NoPngData
This **ccPNG** was default constructed.

- `void pelBuffer(ccPelBuffer<c_UInt16>& pelBuf) const;`

Fills in a pelBuffer representation of this **ccPNG** object.

Parameters

pelBuf The pelBuffer.

Notes

An 8-bit pel buffer (or PNG file) is converted into a 16-bit pel buffer.

A color pel buffer (or PNG file) is converted to greyscale. The algorithm used for converting color PNGs to 8 bit values in a 16-bit image is as follows:
pixel value = (Red + Green + Blue) / 3

pelBuf's image-coordinate offset is read from the data.

pelBuf's client-coordinate transform is set to identity.

The function will discard the originally provided *pelBuf* and allocate an appropriately sized image. In other words, the *pelBuf* provided will be ignored and not be reused.

Throws

ccPNG::NoPngData

This **ccPNG** was default constructed.

- `void pelBuffer(ccPelBuffer<ccPackedRGB32Pel>& pelBuf) const;`

Fills in a color pelBuffer representation of this **ccPNG** object.

Parameters

pelBuf The pelBuffer.

Notes

An 8-bit greyscale pel buffer (or PNG file) is converted to color by copying the greyvalues into all three colors and setting all the alpha values to 0.

A 16-bit pel buffer (or PNG file) is converted to color by first rightshifting by **numSignficantBits()**-8 and then being saturated to 255, and then copying the clamped greyvalues into all three colors and setting all the alpha values to 0.

A 3-plane pel buffer (or PNG file) is converted to a packed 32-bit image by setting all the alpha values to 0.

pelBuf's image-coordinate offset is read from the data.

pelBuf's client-coordinate transform is set to identity.

The function will discard the originally provided *pelBuf* and allocate an appropriately sized image. In other words, the *pelBuf* provided will be ignored and not be reused.

Throws

ccPNG::NoPngData
This **ccPNG** was default constructed.

- `void pelBuffer(cc3PlanePelBuffer& pelBuf) const;`

Fills in a color pelBuffer representation of this **ccPNG** object.

Parameters

pelBuf The pelBuffer.

Notes

An 8-bit greyscale pel buffer (or PNG file) is converted to color by copying the greyvalues into all three planes.

A 16-bit greyscale pel buffer (or PNG file) is converted to color by scaling down to 8 bits by rightshifting by **numSignficantBits()**-8 and then clamping to 255 and then copying into all three planes.

A packed 32-bit color pel buffer (or PNG file) is converted to a color three plane pel buffer by preserving the r,g,b content but discarding the alpha channel.

pelBuf's image-coordinate offset is read from the data.

pelBuf's client-coordinate transform is set to identity.

The function will discard the originally provided *pelBuf* and allocate an appropriately sized image. In other words, the *pelBuf* provided will be ignored and not be reused.

Throws

ccPNG::NoPngData
This **ccPNG** was default constructed.

numSignificantBits

`c_Int32 numSignificantBits() const;`

Gets the number of significant bits.

The *numSignificantBits* is read when loading a PNG file.

The default is 0.

ccPoint

```
#include <ch_cvl/simpshap.h>

class ccPoint : public cc2Vect;
```

Note The **ccPoint** class is deprecated. Use **cc2Point** instead.

ccPoint

ccPointMatcher

```
#include <ch_cvl/ptmatch.h>

class ccPointMatcher;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The Point Matcher tool finds a correspondence between a set of 2D model points (trained points) and a set of 2D data points (runtime points). This correspondence matches each model point with at most one data point and each data point with at most one model point. The number of data points may be more or less than the number of model points. You can assign the model points weights that may be positive, negative, or zero.

The tool finds a transformation (**cc2Xform**) that transforms the matched model points to a set of points that constitutes the best fit of model points among the data points.

The Point Matcher tool currently has two modes:

- Point to point mode: This mode attempts to find a match for model points that may be in any arbitrary position.
- Point to grid mode: This mode attempts to find a match for model points that are known to lie on a grid of 2D points, although each point of the grid may not be a model point. A grid is a set of 2D points of the form:

  ```
  { O + m*U + n*V : m,n are integers }
  ```

  where O is the 2D origin of the grid and U and V are 2D pitch vectors that span the grid.

Caution *The behavior of the Point Matcher tool in point to grid mode is preliminary and subject to change. Further, the point to grid mode currently represents beta level functionality and may not yet have been as extensively tested as the point to point mode.*

Constructors/Destructors

ccPointMatcher `ccPointMatcher();`

Construct a **ccPointMatcher** object.

Public Member Functions

train

```
void train(const cmStd vector<cc2Vect>& modelPoints);

void train(const cmStd vector<cc2Vect>& modelPoints,
  const cmStd vector<c_Int32>& weights);

void train(const ccPelBuffer_const<c_UInt8>& pattern);
```

- `void train(const cmStd vector<cc2Vect>& modelPoints);`

  Trains this **ccPointMatcher** for point to point matching. You can retrain an already trained Point Matcher tool for the same mode or a different mode.

  Parameters

  modelPoints A vector of points.

  Note The weight of each model point is one.

  Figure 1. Class Diagram of ccPointMatcher Class

  Throws

  ccPointMatcherDefs::InsufficientModelPoints

  The *modelPoints* vector contains fewer than three points.

- `void train(const cmStd vector<cc2Vect>& modelPoints, const cmStd vector<c_Int32>& weights);`

  Trains this **ccPointMatcher** for point to point matching with each model point carrying a weight specified by the weights vector. Only the ratio of each weight to the sum of all positive weights is used to calculate coverage for a correspondence.

  You can retrain an already trained Point Matcher tool for the same mode or a different mode.

  Parameters

  modelPoints A vector of points.

weights A vector of weights. Each weight in the vector affects the coverage score as follows:

Weight	Effect on Coverage Score
> 0	A match for this point increases coverage.
0	A match for this point does not affect coverage.
< 0	A match for this point decreases coverage.

Notes

The minimum coverage value for the tool must be less than or equal to the ratio of the number of model points with positive weights to the total number of model points.

Throws

ccPointMatcherDefs::InsufficientModelPoints
The *modelPoints* vector contains fewer than three points.

ccPointMatcherDefs::BadParams
The number of model points is not equal to the number of weights, or there is no positive weight.

- `void train(const ccPelBuffer_const<c_UInt8>& pattern);`

Trains this **ccPointMatcher** for point to grid matching. You can retrain an already trained Point Matcher tool for the same mode or a different mode.

Parameters

pattern Contains information on the placement of points on a grid. The pixel values and **clientFromImageXForm** of the pattern specify model points using the following convention:

A pixel (i,j) in the pattern is non-zero if and only if the point **clientFromImageXForm * cc2Vect(i,j)** is a model point.

Throws

ccPointMatcherDefs::InsufficientModelPoints
The *pattern* contains fewer than three points.

Caution *The behavior of the Point Matcher tool in point to grid mode is preliminary and subject to change. Further, the point to grid mode currently represents beta level functionality and may not yet have been as extensively tested as the point to point mode.*

isTrained `bool isTrained() const;`

Returns true if this **ccPointMatcher** has been trained, false otherwise.

trainMode `ccPointMatcherDefs::Mode trainMode() const;`

Returns the mode for which this **ccPointMatcher** was trained. This function returns one of the following values:

ccPointMatcherDefs::ePointToPoint
ccPointMatcherDefs::ePointToGrid

Throws

ccPointMatcherDefs::NotTrained
This **ccPointMatcher** is not trained for either matching mode.

untrain `void untrain();`

Untrains this **ccPointMatcher**.

Notes

Calling this function has no effect if this **ccPointMatcher** is not trained.

run

```
void run(const cmStd vector<cc2Vect>& dataPoints,
  const ccPointMatcherRunParams& rp,
  ccPointMatcherResultSet& results) const;
```

Runs the Point Matcher tool and stores the results to the results object.

Parameters

dataPoints	The run-time points to match.
rp	A **ccPointMatcherRunParams** object containing the parameters to use for this run.
results	A reference to a **ccPointMatcherResultSet** to which the results of this run are stored.

Notes

The **ccPointMatcher::run()** function supports the use of **ccTimeout**-based time-outs (defined in *ch_cvl/attent.h*). This function terminates if it runs longer than the time-out specified by a user-created **ccTimeout** object. If the function times out, no valid results are produced.

Throws

ccPointMatcherDefs::NotTrained

This **ccPointMatcher** object is not trained for either matching mode.

ccTimeout::Expired

The execution time of this function has exceeded the time-out interval specified by a user-created **ccTimeout** object.

modelPoints

```
const cmStd vector<cc2Vect>& modelPoints() const;
```

Returns the trained points.

Throws

ccPointMatcherDefs::NotTrained

This **ccPointMatcher** object is not trained for point to point matching.

weights

```
cmStd vector<c_Int32> weights() const;
```

Returns the weights vector for model points. Returns a vector of ones if the Point Matcher tool was trained with the **ccPointMatcher::train()** overload that does not take a vector of weights.

Throws

ccPointMatcherDefs::Not Trained

This **ccPointMatcher** object is not trained for point to point matching.

pattern

```
const ccPelBuffer_const<c_UInt8>& pattern() const;
```

Returns the model pattern.

Throws

ccPointMatcherDefs::NotTrained

This **ccPointMatcher** object is not trained for point to grid matching.

ccPointMatcherDefs

```
#include <ch_cvl/ptmatch.h>

class ccPointMatcherDefs;
```

A name space that holds enumerations and constants used with the Point Matcher tool.

Enumerations

Mode

```
enum Mode;
```

This enumeration defines types of point matching supported by the Point Matcher tool.

Value	Meaning
ePointToPoint	Point-to-point matching
ePointToGrid	An optimization of point-to-point matching that assumes that the points lie on a grid

ccPointMatcherDefs

ccPointMatcherResult

```
#include <ch_cvl/ptmatch.h>

class ccPointMatcherResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single Point Matcher tool result.

Constructors/Destructors

ccPointMatcherResult

```
ccPointMatcherResult();
```

Construct **ccPointMatcherResult** with no valid data.

Public Member Functions

runMode

```
ccPointMatcherDefs::Mode runMode() const;
```

Returns the mode in which the tool was run to get this result. This function returns one of the following values:

ccPointMatcherDefs::ePointToPoint
ccPointMatcherDefs::ePointToGrid

Notes

Returns *ccPointMatcherDefs::ePointToPoint* if this **ccPointMatcherResult** is default-constructed.

modelToDataXform

```
const cc2Xform& modelToDataXform() const;
```

Returns a **cc2Xform** object that describes the transformation from the trained points to the run-time points.

Notes

Returns **cc2Xform::I** if this **ccPointMatcherResult** is default-constructed.

coverage `double coverage() const;`

Returns the matching coverage score for this result as a number from 0.0 to 1.0, inclusive. Returns 0.0 if this result object was default constructed.

Note Matching coverage is the larger of zero and the ratio of the sum of the weights of the matched model points to the sum of all positive weights. If all model points have weights of one, coverage is equal to the ratio of the number of matched model points to the total number of model points.

modelToDataMap

```
const cmStd vector<c_Int32>& modelToDataMap() const;
```

Returns a vector that gives the correspondence between trained points and matched run-time points. Each element of the returned vector gives the index of the matched run-time point that corresponds to that trained point.

The size of the returned vector is same as the size of vector of model points. Model points that were not matched are assigned a value of -1.

Notes

Returns a zero-length vector if this **ccPointMatcherResult** is default-constructed.

Throws

ccPointMatcherDefs::NotRun

These results were not obtained using point-to-point mode.

fitError `double fitError() const;`

Returns the point fitting error. This is the RMS error between the transformed matched model points and the corresponding data points. Unmatched points are not considered in computing this score.

Notes

Returns 0.0 if this **ccPointMatcherResult** is default-constructed.

ccPointMatcherResultSet

```
#include <ch_cvl/ptmatch.h>

class ccPointMatcherResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a set of Point Matcher tool results.

Constructors/Destructors

ccPointMatcherResultSet

```
ccPointMatcherResultSet();
```

Construct a result set object

Public Member Functions

time

```
double time() const;
```

Returns the time in seconds required to compute this result set.

results

```
const cmStd vector<ccPointMatcherResult>& results() const;
```

Returns a vector containing all of the results for this run of the tool.

ccPointMatcherResultSet

ccPointMatcherRunParams

```
#include <ch_cvl/ptmatch.h>

class ccPointMatcherRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains Point Matcher tool run-time parameters.

Constructors/Destructors

ccPointMatcherRunParams

```
ccPointMatcherRunParams(double minCoverage=0.5,
  const cc2Xform& startPose=cc2Xform(),
  const ccRange& xTranslationUncertainty =
    ccRange::FullRange(),
  const ccRange& yTranslationUncertainty =
    ccRange::FullRange(),
  const ccAngleRange& rotationUncertainty =
    ccAngleRange(ccDegree(-15), ccDegree(15)),
  const ccRange& scaleUncertainty = ccRange(0.0, 0.0),
  bool autoGridSize = true, double gridSize=0.0,
  bool autoCaptureRange = true, double captureRange=0.0,
  c_Int32 maxNumResults=1);
```

Constructs a **ccPointMatcherRunParams** and initializes it to the supplied values.

Parameters

minCoverage	Only matches where the coverage ratio is greater than or equal to *minCoverage* are evaluated and returned. Matching coverage is the larger of zero and the ratio of the sum of the weights of the matched model points to the sum of all positive weights.
startPose	The initial nominal pose for the run-time points. Each trained point is mapped through *startPose* before the initial point correspondences are performed.

xTranslationUncertainty
: The maximum expected displacement in the x-direction with respect to *startPose*. The tool will attempt to match the run-time points at all points along this range.

yTranslationUncertainty
: The maximum expected displacement in the y-direction with respect to *startPose*. The tool will attempt to match the run-time points at all points along this range.

rotationUncertainty
: The maximum expected rotation with respect to *startPose*. The tool will attempt to match the run-time points across the angular range you specify.

scaleUncertainty
: The maximum expected scale change with respect to *startPose*. The tool will attempt to match the run-time points across the range of scale changes you specify.

autoGridSize
: If true, then the user-specified value of *gridSize* is not used and the tool automatically sets the grid size to be 25% of the distance between the closest pair of points in the trained point set. If false, the user-specified value for *gridSize* is used instead.

gridSize
: The grid size is used to quantize point locations before computing the coarse transformation. The *autoGridSize* parameter must be false for the user-specified value of *gridSize* to be used.

autoCaptureRange
: If true, the tool automatically sets the capture range to be 25% of the distance between the closest pair of points in the trained point set. If false, the user-specified value for *captureRange* is used instead.

captureRange
: The capture range is the radius from each model point where a run-time point is considered to be a matching point. The *autoCaptureRange* parameter must be false for the user-specified value of *captureRange* to be used.

maxNumResults
: The maximum number of results to compute. The tool will not return more than *maxNumResults*. The results are returned in descending coverage score order, and within matching coverage scores in increasing fit error order.

Throws

ccPointMatcherDefs::BadParams

minCoverage is less than 0.0 or greater than 1.0; one or more of the uncertainty ranges is empty; *scaleUncertainty* is full; the start of *scaleUncertainty* range is less than -0.9; *autoGridSize* is false and *gridSize* is less than or equal to 0.0; *autoCaptureRange* is false and *captureRange* is less than 0.0; or *maxNumResults* is less than 1.

Operators

operator==

```
bool operator==(const ccPointMatcherRunParams& that) const;
```

Returns true if this **ccPointMatcherRunParams** is equal to the supplied object, otherwise returns false. Two **ccPointMatcherRunParams** objects are equal if and only if all of their corresponding data members are equal.

Parameters

that The **ccPointMatcherRunParams** to compare to this one.

Public Member Functions

minCoverage

```
void minCoverage(double minCoverage);

double minCoverage() const;
```

- `void minCoverage(double minCoverage);`

Sets the coverage threshold. Only those matches where the ratio of matched points to trained points is greater than or equal to the coverage threshold are marked as found results.

Parameters

minCoverage The coverage threshold. Must be from 0.0 through 1.0.

Notes

Matching coverage is defined to be the larger of zero and the ratio of the sum of the weights of matched model points to the sum of all positive weights. If all model points have weights of one, the coverage is equal to the ratio of the number of matched model points to the total number of model points.

If the point matcher tool is trained with weights, *minCoverage* must be less than or equal to the ratio of the number of model points with positive weights to the total number of model points.

Throws

ccPointMatcherDefs::BadParams

minCoverage is less than 0.0 or greater than 1.0.

- `double minCoverage() const;`

  Returns the coverage threshold. Only those matches where the ratio of matched points to trained points is greater than or equal to the coverage threshold are marked as found results.

startPose

```
void startPose(const cc2Xform& startPose);

cc2Xform startPose() const;
```

- `void startPose(const cc2Xform& startPose);`

  Sets the starting pose for this **ccPointMatcherRunParams**. The starting pose is applied to each trained point before the points are matched.

  Parameters

  startPose The starting pose.

- `cc2Xform startPose() const;`

  Returns the starting pose for this **ccPointMatcherRunParams**.

xTranslationUncertainty

```
void xTranslationUncertainty(
  const ccRange& xTranslationUncertainty);

ccRange xTranslationUncertainty() const;
```

- ```
  void xTranslationUncertainty(
    const ccRange& xTranslationUncertainty);
  ```

  Sets the translation uncertainty in the x-direction. The Point Matcher tool searches for and evaluates point matches within the specified translation range, relative to the starting pose.

  Parameters

  xTranslationUncertainty

  A **ccRange** giving the range of translation to evaluate.
  ```

Throws

ccPointMatcherDefs::BadParams
xTranslationUncertainty is empty.

- `ccRange xTranslationUncertainty() const;`

  Returns the translation uncertainty in the x-direction.

yTranslationUncertainty

```
void yTranslationUncertainty(
 const ccRange& yTranslationUncertainty);

ccRange yTranslationUncertainty() const;
```

- `void yTranslationUncertainty(`
  `const ccRange& yTranslationUncertainty);`

  Sets the translation uncertainty in the y-direction. The Point Matcher tool searches for and evaluates point matches within the translation range, relative to the starting pose.

  Parameters

  yTranslationUncertainty
  A **ccRange** giving the range of translation to evaluate.

  Throws

  ccPointMatcherDefs::BadParams
  yTranslationUncertainty is empty.

- `ccRange yTranslationUncertainty() const;`

  Returns the translation uncertainty in the y-direction.

rotationUncertainty

```
void rotationUncertainty(
 const ccAngleRange& rotationUncertainty);

ccAngleRange rotationUncertainty() const;
```

- `void rotationUncertainty(`
  `const ccAngleRange& rotationUncertainty);`

  Sets the rotational uncertainty. The Point Matcher tool searches for and evaluates point matches within the specified angle range, relative to the starting pose.

Parameters

rotationUncertainty

A **ccAngleRange** giving the range of rotation to evaluate.

Throws

ccPointMatcherDefs::BadParams

rotationUncertainty is empty.

- `ccAngleRange rotationUncertainty() const;`

Returns the rotational uncertainty.

scaleUncertainty

```
void scaleUncertainty(const ccRange& scaleUncertainty);

ccRange scaleUncertainty() const;
```

- `void scaleUncertainty(const ccRange& scaleUncertainty);`

Sets the scale uncertainty. The Point Matcher tool searches for and evaluates point matches within the specified scale range.

Parameters

scaleUncertainty

A **ccRange** giving the range of scale to evaluate.

Throws

ccPointMatcherDefs::BadParams

scaleUncertainty is empty or full or the start of *scaleUncertainty* is less than -0.9.

- `ccRange scaleUncertainty() const;`

Returns the scale uncertainty.

Notes

The start and end values of the supplied **ccRange** are interpreted as the fractional scale changes. For example, setting a range of -0.1 to 0.2 means scales between 90% to 120% relative to the starting pose are evaluated.

autoGridSize

```
void autoGridSize(bool autoGridSize);

bool autoGridSize() const;
```

- `void autoGridSize(bool autoGridSize);`

  Sets whether or not this **ccPointMatcherRunParams** is configured to automatically determine the grid size.

  Parameters

  autoGridSize Set to true to have this **ccPointMatcherRunParams** automatically set the grid size to 25% of the distance between the two closest points in the trained point set, false to use the value supplied to **gridSize()**.

- `bool autoGridSize() const;`

  Returns true if this **ccPointMatcherRunParams** is configured to automatically determine the grid size, false if the grid size supplied to **gridSize()** is used.

gridSize

```
void gridSize(double gridSize);

double gridSize() const;
```

- `void gridSize(double gridSize);`

  Sets the grid size for this **ccPointMatcherRunParams**. The grid size specifies the size of the cells in the grid used to quantize both trained and run-time point locations before computing the coarse transformation. If *autoGridSize* is true, then the grid size set using this setter is ignored.

  You should specify a grid size that is small enough that no more than one point is quantized to a given grid cell.

  Parameters

  gridSize The grid size.

  Throws

  ccPointMatcherDefs::BadParams

  gridSize is less than or equal to 0.0 and this **ccPointMatcherRunParams** is not configured for automatic grid size determination.

Notes

If you have specified automatic grid size determination (by either calling **autoGridSize()** with an argument of true or setting *autoGridSize* to true at construction), the tool automatically sets the grid size. Otherwise, you should specify the grid size to be approximately 25% of the distance between the two trained points that are closest to each other and set *autoGridSize* to false.

- `double gridSize() const;`

  Returns the grid size for this **ccPointMatcherRunParams**.

autoCaptureRange

```
void autoCaptureRange(bool autoCaptureRange);

bool autoCaptureRange() const;
```

- `void autoCaptureRange(bool autoCaptureRange);`

  Sets whether or not this **ccPointMatcherRunParams** is configured to automatically determine the capture range.

  Parameters

  autoCaptureRange

  Set to true to have this **ccPointMatcherRunParams** automatically set the capture range to be 25% of the distance between the closest pair of points in the trained point set, false to use the value supplied to **captureRange()**

- `bool autoCaptureRange() const;`

  Returns true if this **ccPointMatcherRunParams** is configured to automatically determine the capture range, false if the capture range to **captureRange()** is used.

captureRange

```
void captureRange(double captureRange);

double captureRange() const;
```

- `void captureRange(double captureRange);`

  Sets the capture range for this **ccPointMatcherRunParams**. Any run-time point lying within the capture range from a trained point is treated as a matching point. If multiple run-time points lie within the capture range of a single trained point, then a separate result is computed for each point. If *autoCaptureRange* is false, then the capture range set using this setter is ignored.

  Parameters

  captureRange The capture range.

  Throws

  ccPointMatcherDefs::BadParams

  captureRange is less than or equal to 0.0 and this **ccPointMatcherRunParams** is not configured for automatic capture range determination.

  Notes

  If you have specified capture range size determination (by calling **autoCaptureRange()** with an argument of true or by setting *autoCaptureRange* to true at construction), the tool automatically sets the capture range. Otherwise, you should set the capture range to be approximately 25% of the distance between the two trained points that are closest to each other and set *autoCaptureRange* to false.

- `double captureRange() const;`

  Returns the capture range for this **ccPointMatcherRunParams**.

maxNumResults

```
void maxNumResults(c_Int32 maxNumResults);

c_Int32 maxNumResults() const;
```

- `void maxNumResults(c_Int32 maxNumResults);`

  Sets the maximum number of results to compute. The Point Matcher tool will only compute the requested number of matches.

  Parameters

  maxNumResults The maximum number of results. *maxNumResults* must be greater than 0.

Throws

ccPointMatcherDefs::BadParams
maxNumResults is less than 1.

- `c_Int32 maxNumResults() const;`

  Returns the maximum number of results.

ccPointSet

```
#include <ch_cvl/simpshap.h>

class ccPointSet;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes a set of points. **ccPointSet** uses a the C++ Standard Library template class vector to create an array of points.

Note **ccPointSet** is one of the few shapes that does not inherit from **ccShape**.

Constructors/Destructors

ccPointSet

```
ccPointSet();

ccPointSet(cmStd vector<cc2Vect> &pts);
```

- `ccPointSet();`

  The default constructor creates an empty point set.

- `ccPointSet(cmStd vector<cc2Vect> &pts);`

  Creates a point set that contains all the points in the *pts* vector.

  Parameters

pts	A vector of points.

Operators

operator[]

```
cc2Vect& operator[] (long i);
cc2Vect const& operator[] (long i) const;
```

- `cc2Vect& operator[] (long i);`

  Provides array-like access to the point set.

  Parameters

  i The index of the point.

- `cc2Vect const& operator[] (long i) const;`

  Provides array-like access to the point set.

  Parameters

  i The index of the point.

operator==

```
bool operator==(const ccPointSet& that) const;
```

Returns true if this point set is equal to *that*, false otherwise. The comparison tests for equality of the points in the set, taking into account a small tolerance.

Parameters

that The other point set.

operator!=

```
bool operator!=(const ccPointSet& that) const;
```

Returns true if this point set is not equal to *that*, false otherwise. The comparison tests for equality of the points in the set, taking into account a small tolerance.

Parameters

other The other point set.

Public Member Functions

map

```
ccPointSet map(const cc2Xform& c) const;
```

Returns a point set that is the result of mapping this point set with the transformation object *c*.

Parameters

c The transformation object.

mapCentered

```
ccPointSet mapCentered(const cc2Xform& c) const;
```

Returns a point set that is the result of mapping this point set with the transformation object *c* using the first point in the set as the origin.

Parameters

c The transformation object.

pos

```
cc2Vect pos() const;
```

Returns the position of the point set (the first point in the set).

angle

```
ccRadian angle() const;
```

Returns the angle of the point set (the angle between the first two points in the set).

boundingBox

```
ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this set of points. See **ccShape::boundingBox()** for more information.

distanceToPoint

```
double distanceToPoint(const cc2Vect& v) const;
```

Returns the minimum distance from this shape to the given point, computed as the distance between the point *v* and **nearestPoint(v)**.

Parameters

v The point.

size

```
int size() const;
```

Returns the number of points in the set.

degen

```
bool degen() const;
```

Returns true if the point set is degenerate, that is the set contains fewer than 2 points.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint `double distToPoint(const cc2Vect& v) const;`

Use **distanceToPoint()** instead of this function.

ccPolarSamplingParams

```
#include <ch_cvl/polar.h>

class ccPolarSamplingParams;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Yes

This class contains the parameters needed to specify a sampling region for polar transformation. The region is an annular section that is divided into a (polar) grid of smaller sampling areas. The function **cfPolarTransformImage()** maps the source pixels within each sampling area to a corresponding pixel of the destination image.

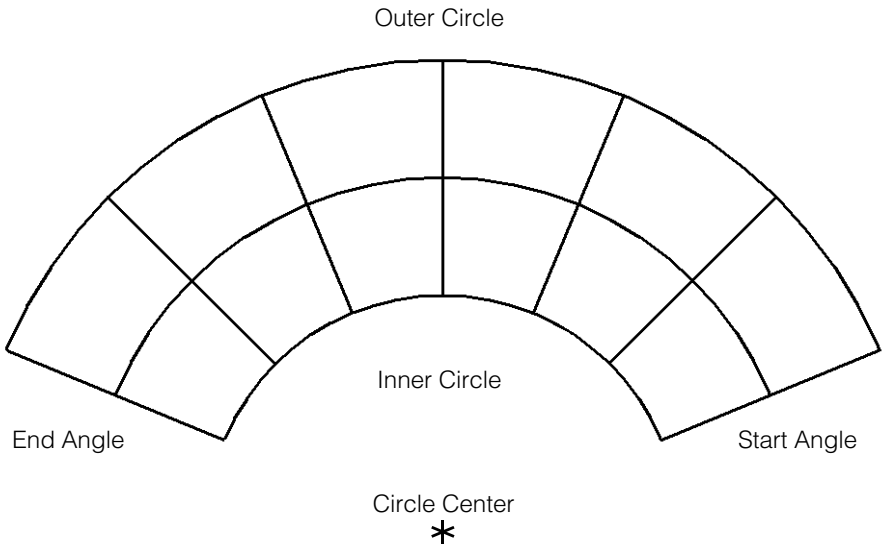
A polar sampling region is specified using the following parameters:

- An ellipse annulus section. This object describes an angular wedge, or section, of the annulus formed between two aligned ellipses with the same major/minor axis ratio.
- The number of X and Y samples. These parameters specify the size of the destination image and (equivalently) the number of divisions in the polar sampling grid. *yNumSamples* nominally specifies the height of the destination image and the number of samples in the radial grid direction. *xNumSamples* nominally specifies the width of the destination and the number of samples in the angular grid direction (see picture below).
- An interpolation method selects how the image pixels are processed to generate the sample value at each sampling point. If you specify no interpolation, the sample value is the value of the single pixel whose center lies nearest to the sampling point. If you specify bilinear interpolation, the sample value is the bilinearly weighted average of the four pixels whose centers are nearest to the sampling point.

The function **cfPolarTransformImage()** maps pixels in the (Cartesian) source image to pixels in the (polar) destination image. Each pixel in the destination image contains some number of whole and/or fractional pieces of the source image pixels. The source pixels are averaged together and normalized to form the corresponding output pixel for the destination image.

See also the Polar Coordinate Transformation Tools section of the Image Transformation Tools chapter in the *CVL Vision Tools Guide*.

The polar transformation tool works in client coordinates.



In the above example, *xNumSamples* = 6 and *yNumSamples* = 2.
There are six angular divisions and two radial divisions.

Constructors/Destructors

ccPolarSamplingParams

```
ccPolarSamplingParams();

ccPolarSamplingParams(
  const ccEllipseAnnulusSection &section,
  c_Int16 xNumSamples, c_Int16 yNumSamples,
  enum ccPolarTransDefs::Interpolation method =
  ccPolarTransDefs::kDefaultInterpolation);

ccPolarSamplingParams(const ccPolarSamplingParams&);
```

- `ccPolarSamplingParams();`

  Default constructor.

- ```
  ccPolarSamplingParams(
    const ccEllipseAnnulusSection &section,
    c_Int16 xNumSamples, c_Int16 yNumSamples,
    enum ccPolarTransDefs::Interpolation method =
    ccPolarTransDefs::kDefaultInterpolation);
  ```

  Convert constructor. Constructs a **ccPolarSamplingParams** object with the given parameters.

  Parameters

section	The **ccEllipseAnnulusSection** that describes the angular wedge, or section, of the annulus formed between two aligned ellipses with the same major/minor axis ratio.
xNumSamples	Number of samples in the horizontal direction.
yNumSamples	Number of samples in the vertical direction.
method	Interpolation method. The default method is **ccPolarTransDefs::kDefaultInterpolation**.

  Throws

  ccPolarTransDefs::BadParams
  The inner ellipse is equal to the outer ellipse.
  The end angle for the section is less than the start angle.
  xNumSamples is less than one.
  yNumSamples is less than one.

- ```
  ccPolarSamplingParams(const ccPolarSamplingParams&);
  ```

  Copy constructor.

Operators

operator=

```
ccPolarSamplingParams& operator=(
  const ccPolarSamplingParams& psp);
```

Assigns the value *psp* to this **ccPolarSamplingParams** object.

Parameters

psp	The **ccPolarSamplingParams** to assign to this one.

operator==

```
bool operator==(const ccPolarSamplingParams& that) const;
```

Returns true if this polar sampling parameters object equals *that*, false otherwise. Two **ccPolarSamplingParams** objects are considered equal if their *section*, *xNumSamples*, *yNumSamples*, and *interpolation method* properties are equal.

Parameters

that The **ccPolarSamplingParams** object to compare to this one.

Public Member Functions

section

```
const ccEllipseAnnulusSection &section() const;

ccEllipseAnnulusSection &section();

void section(const ccEllipseAnnulusSection &section);
```

- `const ccEllipseAnnulusSection &section() const;`

  Retrieves the **ccEllipseAnnulusSection** associated with these polar sampling parameters. The section describes the angular wedge of the annulus formed between two aligned ellipses with the same major/mionor axis ratio.

- `ccEllipseAnnulusSection &section();`

  Retrieves the **ccEllipseAnnulusSection** associated with these polar sampling parameters. The section describes the angular wedge of the annulus formed between two aligned ellipses with the same major/mionor axis ratio. This is the non-const version of the getter.

- `void section(const ccEllipseAnnulusSection &section);`

  Sets the ellipse annulus section associated with these polar sampling parameters. The section describes the angular wedge of the annulus formed between two aligned ellipses with the same major/mionor axis ratio.

  Parameters

  section The **ccEllipseAnnulusSection** to which these polar sampling parameters apply.

  Throws

  ccPolarTransDefs::BadParams
  The inner ellipse of the annulus section is equal to the outer ellipse (**section.innerEllipse() == section.outerEllipse()** evaluates to true).

  ccPolarTransDefs::BadParams
  The end angle for the section is less than the start angle for the section.

xNumSamples

```
c_Int16 xNumSamples() const;
void xNumSamples(c_Int16 xNum);
```

- `c_Int16 xNumSamples() const;`

  Returns the number of samples to be made in the X direction of the destination image. Nominally, this is also the number of divisions of the sampling grid in the angular direction.

- `void xNumSamples(c_Int16 xNum);`

  Sets the number of samples to be made in the X direction of the destination image. Nominally, this is also the number of divisions of the sampling grid in the angular direction.

  Parameters

  xNum The 16-bit integer specifying the number of samples to be made in the X direction of the destination image.

  Throws

  ccPolarTransDefs::BadParams
  xNum is less than one.

yNumSamples

```
c_Int16 yNumSamples() const;
void yNumSamples(c_Int16 yNum);
```

- `c_Int16 yNumSamples() const;`

  Returns the number of samples to be made in the Y direction of the destination image. Nominally, this is also the number of divisions of the sampling grid in the radial direction.

- `void yNumSamples(c_Int16 yNum);`

  Sets the number of samples to be made in the Y direction of the destination image. Nominally, this is also the number of divisions of the sampling grid in the radial direction.

  Parameters

  yNum The 16-bit integer specifying the number of samples to be made in the Y direction of the destination image.

  Throws

  ccPolarTransDefs::BadParams
  yNum is less than one.

interpolation

```
enum ccPolarTransDefs::Interpolation interpolation()
  const;

void interpolation(
  enum ccPolarTransDefs::Interpolation interpolation);
```

- ```
  enum ccPolarTransDefs::Interpolation interpolation()
    const;
  ```

  Returns the interpolation method used during projection. The interpolation method determines how the pixels around the sample point are combined to create the value of the sample.

- ```
  void interpolation(
    enum ccPolarTransDefs::Interpolation interpolation);
  ```

  Sets the interpolation method used during projection. The interpolation method determines how the pixels around the sample point are combined to create the value of the sample.

  Parameters

  interpolation The interpolation method used during projection. Must be one of:

  ccPolarTransDefs::eNone
  ccPolarTransDefs::eBilinear
  ccPolarTransDefs::kDefaultInterpolation

  The default interpolation is bilinear.

willClip

```
bool willClip(const cc_PelBuffer& srcImg) const;
```

Returns true if the polar sampling parameters will access pixels outside of the specified source image.

Parameters

srcImg The source image that the sampling parameters will access.

Throws

ccPel::UnboundWindow
srcImg is an unbound pel buffer.

mapPolarPosition

```
cc2Vect mapPolarPosition(const cc2Vect &polarPosition)
  const;
```

Returns the position in the original source image corresponding to the given position in the (polar transformed) destination image. The source image position is returned in client coordinates. The destination image position is specified in image coordinates.

Parameters

polarPosition Position in the (polar transformed) destination image in image coordinates.

mapImagePosition

```
cc2Vect mapImagePosition(const cc2Vect &imagePosition)
  const;
```

Returns the position in the (polar transformed) destination image corresponding to the given position in the original source image. The destination image position is returned in image coordinates. The source image position is specified in client coordinates.

Parameters

imagePosition Position in the original source image in client coordinates.

Throws

ccPolarTransDefs::BadParams

imagePosition is at the center of the ellipse annulus section associated with these polar sampling parameters. In other words *imagePosition* is equal to **section().center()**.

ccPolarSamplingParams

ccPolarTransDefs

```
#include <ch_cvl/polar.h>

class cmImport_cogip ccPolarTransDefs;
```

A name space class that holds enumerations and errors associated with polar transformations.

Enumerations

Interpolation

```
enum Interpolation;
```

Interpolation methods used for polar transformation.

Value	Meaning
eNone = 1	No interpolation
eBilinear = 2	Bilinear interpolation
kDefaultInterpolation = *eBilinear*	The default interpolation method is bilinear.

See also the Polar Coordinate Transformation Tools section of the Image Transformation Tools chapter in the *CVL Vision Tools Guide*.

ccPolarTransDefs

ccPolygon

```
#include <ch_cvl/oldshape.h>

class ccPolygon;
```

This class is deprecated. Use the **ccPolyline** class instead. See **ccPolyline** on page 2639.

ccPolygon

ccPolyline

```
#include <ch_cvl/shapes.h>

class ccPolyline : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccPolyline** class describes a polyline, a shape made up of two or more vertices connected by line segments. A polyline can be open or closed. Polylines can self-intersect, although some methods in **ccPolyline** and the **ccShape** base class assume non-self-intersecting polylines. These assumptions are documented where applicable.

Constructors/Destructors

ccPolyline

```
explicit ccPolyline(bool closed = false);

ccPolyline(const cmStd vector<cc2Vect> &vertices,
  bool closed = false);
```

- `explicit ccPolyline(bool closed = false);`

  Constructs a **ccPolyline** with the supplied open/closed status. The default constructor constructs an open **ccPolyline** with no vertices. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

  closed — True specifies that the polyline is a closed shape, false that it is open.

  Notes

  The open/closed status of a **ccPolyline** is controlled through the **isClosed()** flag, and is independent of the vertex positions. For instance, a **ccPolyline** may be open even though its first and last vertex coincide or it may be closed when they do not, in which case there is an implied line segment connection between the last and first vertex.

- `ccPolyline(const cmStd vector<cc2Vect> &vertices, bool closed = false);`

  Constructs a **ccPolyline** with an initial set of vertices and the supplied open/closed status. A **ccPolyline** may have any number of vertices, including zero.

  Parameters

  closed — True specifies that the polyline is a closed shape, false that it is open.

Operators

operator==

```
bool operator==(const ccPolyline &rhs) const;
```

Returns true if and only if this **ccPolyline** is equal to *rhs*. Two **ccPolyline** objects are equal if their vertex sequences are identical (no tolerance is used in coordinate comparisons), and they have the same open/closed status.

Parameters

rhs — The other polyline.

operator!=

```
bool operator!=(const ccPolyline &rhs) const;
```

Returns true if *rhs* is not equal to this **ccPolyline**.

Parameters

rhs — The other polyline.

operator=

```
ccPolyline& operator=(const ccPolyline &rhs);
```

Assigns *rhs* to this ccPolyline.

Parameters

rhs — The other polyline.

Public Member Functions

numVertices

```
c_Int32 numVertices() const;
```

Returns the number of vertices of this **ccPolyline**.

vertex

```
const cc2Vect &vertex(c_Int32 idx) const;
void vertex(c_Int32 idx, const cc2Vect &vertex);
```

- `const cc2Vect &vertex(c_Int32 idx) const;`

  Gets the indexed vertex of this **ccPolyline**.

  Parameters

  idx The index.

- `void vertex(c_Int32 idx, const cc2Vect &vertex);`

  Sets the indexed vertex of this **ccPolyline**.

  Parameters

  idx The index.

  vertex The value to assign the indexed vertex.

  Throws

  ccShapesError::BadIndex

  idx is less than zero or greater than or equal to **numVertices()**.

vertices

```
const cmStd vector<cc2Vect> &vertices() const;
void vertices(const cmStd vector<cc2Vect> &vertices);
```

- `const cmStd vector<cc2Vect> &vertices() const;`

  Gets the entire set of vertices of this **ccPolyline**.

- `void vertices(const cmStd vector<cc2Vect> &vertices);`

  Sets the entire set of vertices of this **ccPolyline**.

  Parameters

  vertices The set of vertices.

insertVertex

```
void insertVertex(c_Int32 idx, const cc2Vect &vertex);
```

Inserts a vertex into this **ccPolyline**. The vertex is inserted into the vertex sequence so that after the insertion it has index *idx*.

Parameters

idx The index at which to insert the vertex.
A value of 0 inserts the vertex at the front of the vertex sequence. A value of **numVertices()** inserts it at the end of the vertex sequence.

vertex The vertex.

Notes

Inserting a vertex will change the indices of all existing vertices whose indices prior to the insertion were greater than or equal to *idx*.

Throws

ccShapesError::BadIndex
idx is less than zero or greater than **numVertices()** before insertion.

insertVertices

```
void insertVertices(c_Int32 idx,
  const cmStd vector<cc2Vect> &vertices);
```

Inserts multiple vertices into this **ccPolyline**. The vertices are inserted contiguously into the sequence of vertices so that after the insertion, they have indices *idx* through *idx + vertices.size() - 1*.

Parameters

idx The index at which to insert the vertices.
A value of 0 inserts the vertices at the front of the vertex sequence. A value of **numVertices()** inserts them at the end of the vertex sequence.

vertices The vertices.

Notes

Inserting vertices will change the indices of all existing vertices whose indices prior to the insertion were greater than or equal to *idx*.

Throws

ccShapesError::BadIndex
idx is less than zero or greater than **numVertices()** before insertion.

addVertex

```
void addVertex(const cc2Vect &vertex);
```

Inserts a vertex at the end of this **ccPolyline**'s vertex sequence. Functionally equivalent to invoking **insertVertex()** with *idx* equal to **numVertices()**. See **insertVertex()** for details.

Parameters
vertex The vertex.

addVertices

```
void addVertices(const cmStd vector<cc2Vect> &vertices);
```

Inserts multiple vertices at the end of this **ccPolyline**'s vertex sequence. Functionally equivalent to invoking **insertVertices()** with *idx* equal to **numVertices()**. See **insertVertices()** for details.

Parameters
vertices The vertices.

removeVertex

```
void removeVertex(c_Int32 idx);
```

Removes the indexed vertex of this **ccPolyline**.

Parameters
idx The index.

Notes
Removing a vertex will change the indices of all remaining vertices whose indices prior to the removal were greater than *idx*.

Throws
ccShapesError::BadIndex
idx is less than zero or greater than or equal to **numVertices()** before removal.

isClosed

```
bool isClosed() const;
```

```
void isClosed(bool closed);
```

- `bool isClosed() const;`

  Gets the closed status of this **ccPolyline**.

- `void isClosed(bool closed);`

  Sets the closed status of this **ccPolyline**.

  Parameters
  closed True sets this **ccPolyline** to closed, false sets it to open.

Notes

The setter does not change the vertex sequence. Closed **ccPolyline** objects have an assumed connection between the first and last vertices of the sequence, while open ones do not.

map

```
ccPolyline map(const cc2Xform &xform) const;
```

Returns this **ccPolyline** mapped by *xform*.

Parameters

xform The transformation object.

reserve

```
void reserve(c_Int32 n) const;
```

Preallocates enough memory for this **ccPolyline** to store at least the given number of vertices before additional memory allocations are needed. After a successful invocation of this method, **capacity()** will be no less than the given number.

Parameters

n The number of vertices for which memory is to be allocated.

Throws

cmStd bad_alloc

The required amount of memory cannot be allocated. If this throw occurs, this **ccPolyline** and its capacity are not changed.

Notes

Invoking this method with **capacity()** greater than or equal to *n* has no effect.

This method is the analog of the STL vector method **reserve()**.

capacity

```
c_Int32 capacity() const;
```

Returns the number of vertices that can be stored in this **ccPolyline** before more memory must be allocated. The value returned is never less than **numVertices()**.

Notes

This method is the analog of the STL vector method **capacity()**.

perimeter

```
double perimeter() const;
```

Returns the perimeter of this **ccPolyline**.

Notes

Returns zero if this **ccPolyline** has fewer than two vertices.

perimeter `virtual double perimeter() const;`

Returns the perimeter of this polyline.

Notes
This method returns zero if this polyline has fewer than two vertices.

See **ccShape::perimeter()** for more information.

area `double area() const;`

Returns the area enclosed by this closed **ccPolyline**. This is always non-negative, regardless of this **ccPolyline**'s handedness.

Notes
Returns zero if this **ccPolyline** has fewer than three vertices.

Notes
The return value is undefined for self-intersecting **ccPolyline**s.

Throws
ccShapesError::NotRegion
This **ccPolyline** is not closed.

centerArea `cc2Vect centerArea() const;`

Returns the center of area (centroid) of this closed **ccPolyline**.

Throws
ccShapesError::NotRegion
This **ccPolyline** is not a region.

ccShapesError::DegenerateShape
This **ccPolyline** encloses zero area.

Notes
This method requires this polyline to be non-self-intersecting.

The throw conditions listed above are tested in the given order.

centerArcLength `cc2Vect centerArcLength() const;`

Returns the center of arc length of this **ccPolyline**.

Notes
The center of arc length is where the center of mass would be if mass were evenly distributed along the perimeter of the **ccPolyline** but not over any enclosed area. Unlike **centerArea()**, this measure is defined for both open and closed **ccPolyline**s.

Throws

ccShapesError::EmptyShape
This **ccPolyline** has no vertices.

ccShapesError::DegenerateShape
This **ccPolyline** has zero perimeter.

Notes

The throw conditions listed above are tested in the given order.

center

```
cc2Vect center() const;
```

Returns a reasonable center point for this **ccPolyline**.

Throws

ccShapesError::EmptyShape
This **ccPolyline** has no vertices.

Notes

If this **ccPolyline** is a region and encloses non-zero area, this function returns **centerArea()**. Otherwise, if this **ccPolyline** has non-zero perimeter, this function returns **centerArcLength()**. Otherwise, the vertices are all coincident and if there is at least one vertex, its coordinates are returned.

meanVertex

```
cc2Vect meanVertex() const;
```

Returns the average position of the polyline vertices.

Throws

ccShapesError::EmptyShape
This **ccPolyline** has no vertices.

extremalVertexIndex

```
c_Int32 extremalVertexIndex(const ccRadian &direction)
const;
```

Returns the index of the vertex that has the greatest dot product with a unit vector in the given direction.

Parameters

direction The direction expressed in radians.

Notes

If the extremal vertex is not unique, this function returns the index of one of the extremal vertices.

Throws

ccShapesError::EmptyShape
This **ccPolyline** has no vertices.

closestVertexIndex

```
c_Int32 closestVertexIndex(const cc2Vect &point) const;
```

Returns the index of the vertex closest to the given point.

Parameters

point The point.

Throws

ccShapesError::EmptyShape
This **ccPolyline** has no vertices.

Notes

If the closest vertex is not unique, this function returns the index of one of the closest vertices.

nearestPoints

```
void nearestPoints(const ccPolyline &poly,
 cc2Vect &p1, cc2Vect &p2) const;
```

Returns the nearest two points between this polyline and the given polyline.

Parameters

poly The other polyline.

p1 Return parameter for the nearest point on this polyline.

p2 Return parameter for the nearest point on the other polyline.

Throws

ccShapesError::EmptyShape
Either of the two polylines has no vertices.

Notes

If more than one pair of points have the same nearest distance, this function returns an arbitrary pair among them.

principalMomentsArea

```
void principalMomentsArea(cc2Vect &moments,
  cc2Rigid &polyFromPrincipalXform) const;
```

Computes the principal moments of area of this closed, non empty **ccPolyline**. Also computes the transform from the coordinate system in which the principal moments are computed to the system in which this **ccPolyline**'s vertices are defined. The translational component of the returned transform is always **centerArea()**.

Parameters

moments Return parameter principal moments of area.

polyFromPrincipalXform
Return parameter for the transform that maps from the coordinate system in which the moments are computed to the system in which this polyline's vertices are defined.

Throws

ccShapesError::NotRegion
This **ccPolyline** is not a region.

ccShapesError::DegenerateShape
This **ccPolyline** encloses zero area.

Notes

The throw conditions listed above are tested in the given order.

This method requires this **ccPolyline** to be non-self-intersecting.

The principal axis is the axis about which the moment of area is minimzed. Thus, the first component of the returned moments is always less than or equal to the second component.

If the principal moments are exactly equal, the angle of the principal axes is not well-defined. In this case, the angle of the returned **cc2Rigid** object is arbitrarily chosen as zero. If the principal moments are nearly equal, the angle of the returned **cc2Rigid** object is unstable. This occurs for symmetric shapes such as circles or even squares. There is no such instability in the translational component of the returned **cc2Rigid** object nor in the values of the principal moments.

principalMomentsArcLength

```
void principalMomentsArcLength(cc2Vect &moments,
  cc2Rigid &polyFromPrincipalXform) const;
```

Computes the principal moments of arc length of this **ccPolyline**. Also computes the transform from the coordinate system in which the principal moments are computed to the system in which this **ccPolyline**'s vertices are defined. The translational component of the returned transform is always **centerArcLength()**.

Parameters

moments Return parameter for principal moments of arc length.

polyFromPrincipalXform
Return parameter for the transform that maps from the coordinate system in which the moments are computed to the system in which this polyline's vertices are defined.

Throws

ccShapesError::EmptyShape
This **ccPolyline** has no vertices.

ccShapesError::DegenerateShape
This **ccPolyline** has zero perimeter.

Notes

The throw conditions listed above are tested in the given order.

The principal axis is the axis about which the moment of area is minimized. Thus, the first component of the returned moments is always less than or equal to the second component.

If the principal moments are exactly equal, the angle of the principal axes is not well-defined. In this case, the angle of the returned **cc2Rigid** object is arbitrarily chosen as zero. If the principal moments are nearly equal, the angle of the returned **cc2Rigid** object is unstable. This occurs for symmetric shapes such as circles or even squares. There is no such instability in the translational component of the returned **cc2Rigid** object nor in the values of the principal moments.

areaMoment0 `double areaMoment0() const;`

Computes the zeroth moment of area over this closed **ccPolyline**.

Notes

The **areaMoment*()** functions use Greens's Theorem from vector calculus to convert an integral over an area to an integral over its boundary. This has the side effect of flipping the sign of the result if the boundary is left-handed. For example, even though area (that is, the integral of unity over a polygonal region) is always positive, **areaMoment0()** will return a negative value for left-handed boundaries. This sign can be useful, for example in determining handedness or processing polygonal regions with holes.

All computed moments are zero if this **ccPolyline** has fewer than three vertices.

Throws

ccShapesError::NotRegion
This **ccPolyline** is not closed.

areaMoment1

```
void areaMoment1(double &int_1, double &int_x,
  double &int_y) const;
```

Computes moments of area 0 and 1 over this closed **ccPolyline**.

Parameters

int_1	Integral of the function 1 over the area of the closed polyline.
int_x	Integral of the function x over the area of the closed polyline.
int_y	Integral of the function y over the area of the closed polyline.

Throws

ccShapesError::NotRegion
This **ccPolyline** is not closed.

areaMoment2

```
void areaMoment2(double &int_1, double &int_x,
  double &int_y, double &int_x2, double &int_y2,
  double &int_xy) const;
```

Computes the zeroth, first, and second moments and product of area over this closed **ccPolyline**.

Parameters

int_1	Integral of the function 1 over the area of the closed polyline.
int_x	Integral of the function x over the area of the closed polyline.
int_y	Integral of the function y over the area of the closed polyline.
int_x2	Integral of the function x^2 over the area of the closed polyline.
int_y2	Integral of the function y^2 over the area of the closed polyline.
int_xy	Integral of the function xy over the area of the closed polyline.

Throws

ccShapesError::NotRegion
This **ccPolyline** is not closed.

arcLengthMoment0

```
double arcLengthMoment0() const;
```

Computes the zeroth moment of arc length over this **ccPolyline**.

Notes

All computed moments are zero if this **ccPolyline** has fewer than two vertices.

arcLengthMoment1

```
void arcLengthMoment1(double &int_1, double &int_x,
  double &int_y) const;
```

Computes the zeroth and first moments of arc length over this **ccPolyline**.

Parameters

int_1	Integral of the function 1 over the arc length of the polyline.
int_x	Integral of the function x over the arc length of the polyline.
int_y	Integral of the function y over the arc length of the polyline.

arcLengthMoment2

```
void arcLengthMoment2(double &int_1, double &int_x,
  double &int_y, double &int_x2, double &int_y2,
  double &int_xy) const;
```

Computes the zeroth, first, and second moments and product of arc length over this **ccPolyline**.

Parameters

int_1	Integral of the function 1 over the arc length of the polyline.
int_x	Integral of the function x over the arc length of the polyline.
int_y	Integral of the function y over the arc length of the polyline.
int_x2	Integral of the function x^2 over the arc length of the polyline.
int_y2	Integral of the function y^2 over the arc length of the polyline.
int_xy	Integral of the function xy over the arc length of the polyline.

convexHull

```
ccPolyline convexHull() const;
```

Returns the convex hull of this **ccPolyline**. The convex hull is the smallest closed, convex polygon that encloses this **ccPolyline**. The vertices of the convex hull are always a subset of the vertices of this **ccPolyline**.

Notes

This method always computes a right-handed hull.

It returns an empty, closed polyline if this polyline is empty.

generalWindingAngle

```
ccRadian generalWindingAngle(const cc2Vect &p) const;
```

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve. This implementation is equivalent to the **windingAngle()** method of the **ccShape** base class, with the exception that this method may be applied to any polyline, including one that is closed or empty.

Parameters

p	The start point of the vector *p->t* whose angle is measured as the end point *t* traces the curve.

Notes

If this polyline is empty, this method returns 0 radians.

The winding angle of a closed polyline is always a multiple of 2π radians.

See **ccShape::windingAngle()** for more information.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this polyline.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if and only if this polyline is open and has at least one vertex. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if and only if this polyline is closed and has at least one vertex. A polyline is a region even if it encloses zero area. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For polylines, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if this polyline has no vertices, otherwise false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

This function returns true if the perimeter of this polyline is positive. It returns false if the perimeter is zero. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For polylines, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For polylines, this function always returns true. See **ccShape::reverse()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this polyline.

Throws

ccShapesError::EmptyShape
isEmpty() returns true.

See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this polyline to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p The point.

Throws

ccShapesError::EmptyShape
isEmpty() returns true.

See **ccShape::nearestPoint()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this polyline to the given point, as determined by **nearestPoint()**.

Parameters

info Shape information for this polyline.

point The point.

See **ccShape::nearestPerimPos()** for more information.

startPoint

```
virtual cc2Vect startPoint() const;
```

Returns the starting point of this polyline if it is an open contour.

Throws

ccShapesError::NotOpenContour
This polyline is not an open contour.

See **ccShape::startPoint()** for more information.

endPoint

```
virtual cc2Vect endPoint() const;
```

Returns the ending point of this polyline if it is an open contour.

Throws

ccShapesError::NotOpenContour
This polyline is not an open contour.

See **ccShape::endPoint()** for more information.

startAngle

```
virtual ccRadian startAngle() const;
```

Returns the starting tangent direction of this polyline if it is an open contour.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this polyline.

ccShapesError::NoTangent
hasTangent() is false for this polyline.

See **ccShape::startAngle()** for more information.

endAngle

```
virtual ccRadian endAngle() const;
```

Returns the ending tangent direction of this polyline if it is an open contour. Note that polylines can be either open or closed.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this polyline.

ccShapesError::NoTangent
hasTangent() is false for this polyline.

See **ccShape::endAngle()** for more information.

tangentRotation

```
virtual ccRadian tangentRotation() const;
```

Returns the net signed angle through which the tangent vector rotates from the start point to the end point.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this polyline.

ccShapesError::NoTangent
hasTangent() is false for this polyline.

See **ccShape::tangentRotation()** for more information.

windingAngle

```
virtual ccRadian windingAngle(const cc2Vect &p) const;
```

Returns the net signed angle through which the vector *p->t* rotates as *t* traces the curve.

Parameters

p	The start point of the vector *p->t* whose angle is measured as the end point *t* traces this polyline.

Notes

This method is used for determining whether *p* is inside or outside of a region whose boundary includes this polyline.

Throws

ccShapesError::NotOpenContour
isOpenContour() is false for this polyline.

See **ccShape::windingAngle()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Implements the **within()** method of the **ccShape** base class for polylines.

Parameters

p	The point.

Throws

ccShapesError::NotRegion
This polyline is not a region.

See **ccShape::within()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

Returns true if and only if this polyline is right-handed.

Throws

ccShapesError::NotRegion
This polyline is not a region.

Notes

The value returned by this function is undefined for self-intersecting polylines.

See **ccShape::isRightHanded()** for more information.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this polyline. See **ccShape::reverse()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
 ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.
result	Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores polylines for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this polyline mapped by *X*.

Parameters

X The transformation object.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccContourTree** consisting of connected **ccLineSeg**s.

See **ccShape::decompose()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this polyline over the given perimeter range.

Parameters

info Shape information for this polyline.

range The perimeter range.

See **ccShape::subShape()** for more information.

ccPolyline

ccPtrHandle

```
#include <ch_cvl/handle.h>

template <class T,
          class B = ccPtrHandleBase<T>,
          class C = ccPtrHandleBase_const<T> >
  class ccPtrHandle : public B,
                      public virtual ccPtrHandle_const<T,C> ;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

ccPtrHandle is a template class used to create a pointer handle to another class.

T	The class for which a handle pointer is being created.
B	The class that serves as a pointer handle to the base class.
C	The class that serves as a **const** pointer handle to the base class.

For more information on pointer handles, see the section *CVL Programming Conventions* in the chapter *CVL Programming Overview* in the *CVL User's Guide*.

To define a pointer handle **ccAlphaPtrh** to objects of class **ccAlpha**, you would use the following declaration:

```
typedef ccPtrHandle<ccAlpha> ccAlphaPtrh;
```

Note that **ccAlpha** must be derived from **ccRepBase** as shown on page 2727.

To define pointer handles to a class, **ccBeta**, that is derived from a class that is already derived from **ccRepBase**, use the **cmDerivedPtrHdlDcl** macro like this:

1. Define **const** and non-**const** pointer handle types for the ancestor class **ccAlpha**.

   ```
   typedef ccPtrHandle<ccAlpha> ccAlphaPtrh;
   typedef ccPtrHandle_const<ccAlpha> ccAlphaPtrh_const;
   ```

2. Use the **cmDerivedPtrHdlDcl** macro to declare the pointer handle types for **ccBeta**.

   ```
   cmDerivedPtrHdlDcl(ccBetaPtrh, ccBeta,
                      ccAlphaPtrh, ccAlphaPtrh_const);
   ```

The macro defines the pointer handle types **ccBetaPtrh** and **ccBetaPtrh_const** that point the rep class **ccBeta**.

Constructors/Destructors

ccPtrHandle

```
ccPtrHandle(T* rep = 0);
```

Creates a pointer handle to object *rep*. The reference count of *rep* is incremented.

Parameters

rep A pointer to the handled object.

operator=

```
ccPtrHandle& operator=(T*rep);
ccPtrHandle& operator=(const ccPtrHandle& rhs);
```

- `ccPtrHandle& operator=(T*rep);`

Assignment operator. This pointer handle will point to a handled object *rep*. If this pointer handle pointed to an object before assignment, its reference count is decremented. The reference count of *rep* is incremented.

Parameters

rep A pointer to an object of the handled class.

- `ccPtrHandle& operator=(const ccPtrHandle& rhs);`

Assignment operator. This pointer handle will point to the same object that the pointer handle *rhs* points to. If this pointer handle pointed to an object before assignment, its reference count is decremented. The reference count of *rep* is incremented.

Parameters

rhs A pointer handle.

operator*

```
T& operator*() const;
```

Returns a reference to the object that this pointer handle points to.

Throws

ccBadPointer The pointer handle is null.

operator->

```
T* operator->() const;
```

Returns a pointer to the handled object that can be used to refer to data members. This function allows you to use the **->** operator as if the pointer handle were a regular pointer.

Throws

ccBadPointer The pointer handle is null.

operator const void*

```
operator const void* () const;
```

Allows for testing for null pointers. If this pointer handle is null, this function returns a null pointer. Otherwise, it returns a non-null pointer.

Use this cast operator only for testing whether a pointer handle is null. To get the object that the pointer handle points to, use **rep()** instead.

operator!

```
bool operator!() const;
```

Allows testing for null pointers. Returns true if the pointer handle is null.

Public Member Functions

rep

```
T* rep() const;
```

Returns the handled object. Returns null if this pointer handle is null.

disconnectRep

```
T* disconnectRep();
```

Returns the handled object and disconnects it from this handle. The handled object's reference count is decremented, and the pointer handle is set to null.

Macros

cmDerivedPtrHdlDcl

```
cmDerivedPtrHdlDcl(derivedHdl, derivedRep,
  baseHdl, baseHdl_const);
```

Defines pointer handle types for classes that are derived from classes that are derived from **ccRepBase**. This macro defines two types named ***derivedHdl*** and ***derivedHdl*_const**.

For an example of how to use this macro see the beginning of this chapter.

Parameters

derivedHdl The non-**const** type name of the new pointer handle type. Usually ***derivedRep*PtrH**

derivedRep	The name of the class for which to create pointer handle types. This is a class that is derived from a class derived from **ccRepBase**.
baseHdl	The name of the non-**const** pointer handle type to the class from which *derivedRep* is derived.
baseHdl_const	The name of the **const** pointer handle type to the class from which *derivedRep* is derived.

ccPtrHandle_const

```
#include <ch_cvl/handle.h>

template <class T,
          class B = ccPtrHandleBase_const<T> >
  class ccPtrHandle_const : public virtual B ;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

ccPtrHandle_const is a template class that creates a const pointer handle to another class.

T	The class for which a handle pointer is being created.
B	The class that serves as a **const** pointer handle to the base class.

A **const** pointer handle has similar semantics to a C++ **const** pointer. For more information on pointer handles, see the section *CVL Programming Conventions* in the chapter *CVL Programming Overview* in the *CVL User's Guide*.

To define a **const** pointer handle **ccAlphaPtrh_const** to objects of class **ccAlpha**, you would use the following declaration:

```
typedef ccPtrHandle_const<ccAlpha> ccAlphaPtrh_const;
```

Note that **ccAlpha** must be derived from **ccRepBase** as shown on page 2727.

To define pointer handles to a class, **ccBeta**, that is derived from a class that is already derived from **ccRepBase**, use the **cmDerivedPtrHdlDcl** macro like this:

1. Define **const** and non-**const** pointer handle types for the ancestor class **ccAlpha**.

   ```
   typedef ccPtrHandle<ccAlpha> ccAlphaPtrh;
   typedef ccPtrHandle_const<ccAlpha> ccAlphaPtrh_const;
   ```

2. Use the **cmDerivedPtrHdlDcl** macro to declare the pointer handle types for **ccBeta**.

   ```
   cmDerivedPtrHdlDcl(ccBetaPtrh, ccBeta,
                      ccAlphaPtrh, ccAlphaPtrh_const);
   ```

   The macro defines the pointer handle types **ccBetaPtrh** and **ccBetaPtrh_const** that point the rep class **ccBeta**.

Constructors/Destructors

ccPtrHandle_const

```
ccPtrHandle_const(T* rep = 0);
```

Creates a pointer handle to object *rep*. The reference count of *rep* is incremented.

Parameters

rep	A pointer to the handled object.

Operators

operator=

```
ccPtrHandle_const& operator=(T*rep);

ccPtrHandle_const& operator=(const ccPtrHandle_const&
rhs);
```

- ```
  ccPtrHandle_const& operator=(T* rep);
  ```

  Assignment operator. This pointer handle will point to a handled object *rep*. If this pointer handle pointed to an object before assignment, its reference count is decremented. The reference count of *rep* is incremented.

  Parameters

rep	A pointer to an object of the handled class.

- ```
  ccPtrHandle_const& operator=(
    const ccPtrHandle_const& rhs);
  ```

  Assignment operator. This pointer handle will point to the same object that the pointer handle *rhs* points to. If this pointer handle pointed to an object before assignment, its reference count is decremented. The reference count of *rep* is incremented.

  Parameters

rhs	A pointer handle.

operator*

```
const T& operator*() const;
```

Returns a reference to the object that this pointer handle points to.

Throws

ccBadPointer	The pointer handle is null.

operator->

```
const T* operator->() const;
```

Returns a pointer to the handled object that can be used to refer to data members. This function allows you to use the **->** operator as if the pointer handle were a regular pointer.

Throws

ccBadPointer The pointer handle is null.

operator const void*

```
operator const void* () const;
```

Allows for testing for null pointers. If this pointer handle is null, this function returns a null pointer. Otherwise, it returns a non-null pointer.

Use this cast operator only for testing whether a pointer handle is null. To get the object that the pointer handle points to, use **rep()** instead.

operator!

```
bool operator!() const;
```

Allows testing for null pointers. Returns true if the pointer handle is null.

Public Member Functions

rep

```
const T* rep() const;
```

Returns the handled object. Returns null if this pointer handle is null.

disconnectRep

```
const T* disconnectRep();
```

Returns the handled object and disconnects it from this handle. The handled object's reference count is decremented, and the pointer handle is set to null.

ccPtrHandle_const

ccPVEReceiver

```
#include <ch_cvl/pve_xfer.h>

class ccPVEReceiver;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

Programmable Vision Engines (PVE) is a legacy technology from Cognex. This class provides functions to read files written by the following PVE functions, written by PVE versions 5.2 through 8.1:

- **ctr_transmit_model()**
- **cip_transmit_image()**
- **cnls_transmit_model()**

Constructors/Destructors

ccPVEReceiver

```
ccPVEReceiver();

ccPVEReceiver(const ccCvlString& pathname);

ccPVEReceiver(cmStd istream& i,
  const ccCvlString& name = cmT("[unknown]"));
```

- `ccPVEReceiver();`

  Creates an unbound **ccPVEReceiver**.

- `ccPVEReceiver(const ccCvlString& pathname);`

  Creates a **ccPVEReceiver** from a file on disk. If successful, the receiver is bound to the file. This function expects that only one object is stored in a file.

  This constructor is intended for use with temporary instantiations of **ccPVEReceiver**. For example:

```
ccPelBuffer<c_UInt8> pelbuf; // some pel buffer

try {
   // extract an image
  pelbuf = ccPVEReceiver(filename).image();
  ...
}
catch(ccPVEReceiver::Warning& w) { ... }
catch(ccException& e) { ... }
```

  Parameters

  pathname The name of the file.

  Throws

  ccPVEReceiver::FileNotFound
  pathname cannot be opened.

  ccPVEReceiver::BadFormat
  The PVE transfer file is improperly formatted and cannot be read.

  ccPVEReceiver::NotImplemented
  The PVE transfer file contains a valid object type but one that **ccPVEReceiver** cannot parse.

  The PVE transfer file contains an image deeper than 8 bits.

  ccPVEReceiver::Warning
  Initialization encountered a recoverable error. See **warnings()** on page 2674.

- `ccPVEReceiver(cmStd istream& i,`
  `const ccCvlString& name = cmT("[unknown]"));`

  Creates a **ccPVEReceiver** using a stream. If successful, the receiver is bound to the stream.

  Parameters

  i The input stream.

name The name of the stream. This name is used only when generating error messages. It should be the name of the file or input source of the input stream.

Throws

ccPVEReceiver::BadFormat
The PVE transfer stream is improperly formatted and cannot be read.

ccPVEReceiver::NotImplemented
The PVE transfer stream contains a valid object type but one that **ccPVEReceiver** cannot parse.

The PVE transfer file contains an image deeper than 8 bits.

ccPVEReceiver::Warning
Initialization encountered a recoverable error. See **warnings()** on page 2674.

Enumerations

ObjectType

```
enum ObjectType;
```

This enumeration is used by **objectType()** to return the type of object that a **ccPVEReceiver** is bound to.

Value	Meaning
eModel	The object is a PVE model.
eImage	The object is a PVE image or CNLSearch model. (See **isCnls()** on page 2672.)

Public Member Functions

init

```
void init();

void init(const ccCvlString& pathname);

void init(cmStd istream& i,
  const ccCvlString& name = cmT("[unknown]"));
```

- `void init();`

  Unbinds this **ccPVEReceiver** if it was bound.

- `void init(const ccCvlString& pathname);`

  Initializes an existing **ccPVEReceiver** to use a file on disk. If successful, the **ccPVEReceiver** is bound to the file.

  Parameters

  pathname The name of the file.

  Throws

  ccPVEReceiver::FileNotFound
  pathname cannot be opened.

  ccPVEReceiver::BadFormat
  The PVE transfer file is improperly formatted and cannot be read.

  ccPVEReceiver::NotImplemented
  The PVE transfer file contains a valid object type but one that **ccPVEReceiver** cannot parse.

  The PVE transfer file contains an image deeper than 8 bits.

  ccPVEReceiver::Warning
  Initialization encountered a recoverable error. **warnings()** on page 2674.

- ```
  void init(cmStd istream& i,
    const ccCvlString& name = cmT("[unknown]"));
  ```

  Initializes a **ccPVEReceiver** from a stream. If successful, the receiver is bound to the stream, and leaves the stream at the end of the first persisted object. Successive calls to this version of **init()** provide access to other persisted objects in the stream.

  Parameters

  i The input stream.

  name The name of the stream. This name is used only when generating error messages. It should be the name of the file or input source of the input stream.

  Throws

  ccPVEReceiver::BadFormat
  The PVE transfer stream is improperly formatted and cannot be read.

  ccPVEReceiver::NotImplemented
  The PVE transfer stream contains a valid object type but one that **ccPVEReceiver** cannot parse.

  The PVE transfer stream contains an image deeper than 8 bits.
  ```

ccPVEReceiver::Warning
Initialization encountered a recoverable error. **warnings()** on page 2674

image

```
ccPelBuffer_const<c_UInt8> image() const;
```

Returns the image stored in the PVE transfer file or stream bound to this **ccPVEReceiver**.

Throws

ccPVEReceiver::Unbound
This **ccPVEReceiver** has not been successfully initialized.

maskImage

```
ccPelBuffer_const<c_UInt8> maskImage() const;
```

Returns the mask image of the search model stored in the PVE transfer file or stream bound to this **ccPVEReceiver**. If the persisted object was a model that was trained without a mask image, **maskImage()** returns an unbound **ccPelBuffer**.

In mask images, the pixel values have the following meanings:

Pixel value	PVE Type	Description
0	*cse_normal*	An ordinary “care” pixel
1	*cse_border*	A “care” pixel that borders an EdgeType
2	*cse_edge*	An “edge don‘t care” pixel
3	*cse_mask*	A “mask don’t care” pixel

Pixel values 0 and 1 are generally “care” pixels, and pixel values 2 and 3 are “don’t care” pixels. For more information, see the PVE search documentation on the subject of mask images for training and the header file *cse_mdl.h*.

Throws

ccPVEReceiver::Unbound
The **ccPVEReceiver** has not been successfully initialized.

ccPVEReceiver::WrongType
This type of transmitted PVE object has no mask image associated with it (for example, a **cip_buffer** or CNLSearch model).

origin `cc2Vect origin() const;`

Returns the origin of the search model.

Throws

ccPVEReceiver::Unbound
This **ccPVEReceiver** has not been successfully initialized.

ccPVEReceiver::WrongType
objectType() returned *ccPVEReceiver::eImage* and **isCnls()** returned false.

hasID `bool hasID() const;`

Returns whether an ID was part of the PVE object.

id `c_Int32 id() const;`

Returns the ID of the CNLPAS model.

Throws

ccPVEReceiver::WrongType
If **hasID()** is false, that is, if this type of transmitted PVE object has no ID.

isBound `bool isBound() const;`

Returns true if this **ccPVEReceiver** is bound to a file or stream, false otherwise.

objectType `ObjectType objectType() const;`

Returns the object type that this **ccPVEReceiver** contains:

- *ccPVEReceiver::eModel*
- *ccPVEReceiver::eImage*

Notes

The **objectType()** of a CNLSearch model is *ccPVEReceiver::eImage*. To determine if this object contains a CNLSearch model, use **isCnls()**.

isCnls `bool isCnls() const;`

Returns true if the image is a PVE CNLSearch model, false if it is a **cip_buffer**.

imageDepth

```
int imageDepth() const;
```

Returns the depth of the image, **cip_buffer.depth**.

version

```
int version() const;
```

Returns the version number of the PVE object.

filename

```
ccCvlString filename() const;
```

Returns the name of the file that this object is stored in.

Throws

ccPVEReceiver::Unbound
This **ccPVEReceiver** has not been successfully initialized.

modelResolution

```
ccIPair modelResolution() const;
```

Returns the amount by which the original image was subsampled before a model was trained as a pair of integers: **ccIPair(cse_model.prx, cse_model.pry)**.

If these two values are not both equal to 1, then **image()** will return an image that is of reduced resolution (subsampled) compared to the original image upon which the model was trained. See the Search PVE documentation for more information on *prx* and *pry*.

Throws

ccPVEReceiver::Unbound
This **ccPVEReceiver** has not been successfully initialized.

ccPVEReceiver::WrongType
The **ccPVEReceiver** does not contain a model object.

warnings `const ccCvlString& warnings() const;`

Returns any recoverable errors, such as corrupted data, internal inconsistencies, or known PVE bugs, encountered while initializing this **ccPVEReceiver**. If there are no warnings, this function returns a zero-length string.

In general, when any of the **ccPVEReceiver** functions throw an exception during initialization, an unrecoverable error has occurred, and the **ccPVEReceiver** is not bound to a file or stream. However, if a function throws the exception *ccPVEReceiver::Warning* during initialization, a recoverable error has occurred and the persisted object data may still be partially or completely intact.

In the case of an initialization warning, all functions will return correct values and the image will be the correct size, but some or all of its pixels may be corrupted or lost. Such pixels are set to zero. This corruption can happen in the following cases:

- The input serialized object is incomplete.
- Some of the pixels are corrupted because of one of the known bugs in PVE persistence
- One of the error-detection mechanisms (blocking and checksumming) detects data corruption

Use this function or **ccPVEReceiver::Warning.message()** to get an explanation of the warning.

Throws

ccPVEReceiver::Unbound
This **ccPVEReceiver** has not been successfully initialized.

ccRadian

```
#include <ch_cvl/units.h>

class ccRadian;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class describes an angle in radians using a floating point representation.

Note that you can use the constructors to convert from one angle representation to another. For example, to specify 36° in radians, you could write:

```
ccRadian rad(ccDegree(36.0));
```

Constructors/Destructors

ccRadian

```
ccRadian();
ccRadian(ccDegree a);
ccRadian(ccAngle8 a);
ccRadian(ccAngle16 a);
explicit ccRadian(double a);
```

- `ccRadian();`

  Creates an uninitialized **ccRadian** object.

- `ccRadian(ccDegree a);`

  Creates a **ccRadian** object from the given **ccDegree** object.

  Parameters

  a The **ccDegree** object to convert to radians.

- `ccRadian(ccAngle8 a);`

  Creates a **ccRadian** object from the given **ccAngle8** object.

  Parameters

  a The **ccAngle8** object to convert to radians.

- `ccRadian(ccAngle16 a);`

  Creates a **ccRadian** object from the given **ccAngle16** object.

  Parameters

  a The **ccAngle16** object to convert to radians.

- `explicit ccRadian(double a);`

  Creates a **ccRadian** object with the specified value.

  Parameters

  a The angle in radians.

Operators

operator+

```
ccRadian operator+(ccRadian a) const;
```

Returns the result of adding the angle *a* in radians to this angle.

Parameters

a The angle to add to this angle.

operator-

```
ccRadian operator-(ccRadian a) const;
ccRadian operator-() const;
```

- `ccRadian operator-(ccRadian a) const;`

  Returns the result of subtracting the angle *a* in radians from this angle.

  Parameters

  a The angle to subtract from this angle.

- `ccRadian operator-() const;`

  Returns the negative of this angle. The unary minus operator.

operator*

```
ccRadian operator*(double a) const;
double operator*(ccRadian a) const;
friend ccRadian operator*(double a, ccRadian b);
```

- `ccRadian operator*(double a) const;`

  Returns the result of multiplying this angle by *a*.

  Parameters

  a The amount to multiply by.

- `double operator*(ccRadian a) const;`

  Returns the result of multiplying this angle by the angle *a* in radians.

  Parameters

  a The angle to multiply by.

- `friend ccRadian operator*(double a, ccRadian b);`

  Returns the result of multiplying the angle *b* by *a*.

  Parameters

  a The amount to multiply by.

  b The angle to multiply.

operator/

```
ccRadian operator/(double a) const;
double operator/(ccRadian a) const;
```

- `ccRadian operator/(double a) const;`

  Returns the result of dividing this angle by *a*.

  Parameters

  a The amount to divide by.

- `double operator/(ccRadian a) const;`

  Returns the result of dividing this angle by the angle *a* in radians.

  Parameters

  a The angle to divide by.

operator=

```
ccRadian& operator=(double a);
```

Assigns the value *a* to this angle.

Parameters

a The value to assign.

---

operator*=

```
ccRadian& operator*=(double a);
ccRadian& operator*=(ccRadian a);
```

- ```
  ccRadian& operator*=(double a);
  ```

  Multiplies this angle by *a* and returns the result.

  Parameters

  a The value to multiply by.

- ```
  ccRadian& operator*=(ccRadian a);
  ```

  Multiplies this angle by the angle *a* and returns the result.

  Parameters

  a The angle to multiply by.

---

operator/=

```
ccRadian& operator/=(double a);
ccRadian& operator/=(ccRadian a);
```

- ```
  ccRadian& operator/=(double a);
  ```

  Divides this angle by the value *a* and returns the result.

  Parameters

  a The value to divide by.

- ```
  ccRadian& operator/=(ccRadian a);
  ```

  Divides this angle by the angle *a* and returns the result.

  Parameters

  a The angle to divide by.

operator+= `ccRadian& operator+=(ccRadian a);`

Adds the angle *a* to this angle and returns the result.

Parameters

a The angle to add.

operator-= `ccRadian& operator-=(ccRadian a);`

Subtracts the angle *a* from this angle and returns the result.

Parameters

a The angle to subtract.

operator== `bool operator==(ccRadian a) const;`

Returns true if this angle is exactly equal to the angle *a*. See **cfRealEq()** on page 3827 for an equality function that lets you specify a tolerance.

Parameters

a The other angle.

operator!= `bool operator!=(ccRadian a) const;`

Returns true if this angle is not equal to the angle *a*.

Parameters

a The other angle.

operator< `bool operator<(ccRadian a) const;`

Returns true if this angle is less than angle *a*.

Parameters

a The other angle.

operator<= `bool operator<=(ccRadian a) const;`

Returns true if this angle is less than or equal to angle *a*.

Parameters

a The other angle.

operator> `bool operator>(ccRadian a) const;`

Returns true if this angle is greater than angle *a*.

Parameters

a The other angle.

operator>= `bool operator>=(ccRadian a) const;`

Returns true if this angle is greater than or equal to angle *a*.

Parameters

a The other angle.

Public Member Functions

toDouble `double toDouble() const;`

Returns this angle as a **double** value.

plain `double plain() const;`

Returns this angle as a **double**.

norm `ccRadian norm() const;`

Returns this angle normalized to the range from 0 up to (but not including) 2π.

signedNorm `ccRadian signedNorm() const;`

Returns this angle normalized to the range from $-\pi$ up to (but not including) π.

ccRange

```
#include <ch_cvl/range.h>

class ccRange;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class characterizes ranges of values and provides methods for performing the following operations:

- Check whether a value is within a range.
- Create a new range by intersecting one range with another.
- Dilate ranges (Minkowski sum operation).
- Erode ranges (Minkowski difference operation).

For example, the class **ccRange** can specify a range such as the range from 2 through 3, which includes the values 2 and 3 and every value in between (2 <= x <= 3).

Constructors/Destructors

ccRange

```
ccRange ();

ccRange (double start, double end);
```

- `ccRange ();`

  Returns the default constructed empty range.

- `ccRange (double start, double end);`

  Returns a **ccRange** object characterized by the given start and end values.

  Parameters

start	The start position of the range.
end	The end position of the range.

Throws

ccRangeDefs::BadParams
The start is greater than the end.

Public Member Functions

dilate

```
ccRange dilate(const ccRange &r) const;
```

Returns a new **ccRange** object that is created by dilating this range with another range specified by *r*. Dilation is accomplished by using the Minkowski sum of two sets A and B. The resulting range is the set of all points which can be generated by adding an element of A to an element of B.

Parameters

r The range used to dilate this range.

EmptyRange

```
static ccRange EmptyRange();
```

Returns the default constructed empty range.

end

```
double end() const;
```

Returns the end position of a partial range.

Throws

ccRangeDefs::NotPartialRange
This is not a partial range.

erode

```
ccRange erode(const ccRange &r) const;
```

Returns a new **ccRange** object that is created by eroding this range by another range specified by *r*. Erosion is accomplished by using the Minkowski difference between two sets A and B. The resulting range is the set of all values generated by subtracting an element of B from an element of A.

Parameters

r The range used to erode this range.

FullRange

```
static ccRange FullRange();
```

Returns the default constructed full range.

intersect

```
ccRange intersect(const ccRange &r) const;
```

Returns a new **ccRange** object that specifies the intersection of this range and another range specified by *r*.

Parameters

r The range that intersects this range.

isWithin

```
bool isWithin(double x) const;
```

Returns a bool that indicates whether or not the value *x* is inside this range.

Parameters

x The value that may or may not be inside this range.

length

```
double length() const;
```

Returns the length of the range which is defined as one of the following:

DBL_MAX if the range is full (range_ == *eFull*)

0 if the range is empty (range_ == *eEmpty*)

*end_-start_* if the range contains some members (range_ == ePartial).

middle

```
double middle() const;
```

Returns the middle of the given range.

Throws

ccRangeDefs::NotPartialRange
This is not a partial range.

scale

```
ccRange scale(double x) const;
```

Returns a new **ccRange** object constructed by scaling this range by *x*.

Parameters

x The scaling factor that is used to scale this range.

Throws

ccRangeDefs::BadParams
x is less than or equal to 0.

someOverlap

```
bool someOverlap(const ccRange &r,
  double threshold=0) const;
```

Returns a bool that indicates whether two ranges have some overlap.

Parameters

r The range that is compared to this range.

threshold The length of the range that is compared to this range. The length should be greater than 0.

start

```
double start() const;
```

Returns the start position of a partial range.

Throws

ccRangeDefs::NotPartialRange
This is not a partial range.

translate

```
ccRange translate(double x) const;
```

Returns a new **ccRange** object constructed by translating this range by *x*.

Parameters

x The amount by which this range is translated.

Operators

operator==

```
bool operator== (const ccRange& that) const;
```

True if this range equals the other range; false otherwise.

Parameters

that The other range.

Notes

Two **ccRange** objects are considered equal if their range types, start values, and end values are equal or if they are both empty or both full.

operator!=

```
bool operator!= (const ccRange& that) const;
```

True if this range does not equal that range; false if this range equals that range.

Parameters

that The other range.

Notes

Two **ccRange** objects are considered equal if their ranges, start values and end values are equal or if they are both empty or both full.

ccRange

ccRangeDefs

```
#include <ch_cvl/range.h>

class ccRangeDefs;
```

A name space that holds enumerations and exceptions used with range classes.

Enumerations

Range

```
enum Range;
```

Values for the *Range* parameter.

Value	Meaning
ePartial = 0	Range spans a finite segment that begins with the start number and ends with the end number.
eEmpty = 1	Range is completely empty.
eFull = 2	Range is completely full.

ccRangeDefs

ccRasterizationDefs

```
#include <ch_cvl/raster.h>

class ccRasterizationDefs;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	Simple

This class serves as a namespace for the sampling parameters passed to **ccShape::sample()** operations called by the **cfRasterize()** and **cfRasterizeContour()** global functions. These functions are used to rasterize regions and contours defined by **ccShape** objects.

Note The default constructor is never called. This class has no instances.

Enumerations

BoundaryFillMode

```
enum BoundaryFillMode {
  eUseInterpolatedPelValue,
  eUseLargestPortionPelValue,
  eUseForegroundPelValue,
  eUseBackgroundPelValue,
  kDefaultBoundaryFillMode = eUseInterpolatedPelValue
};
```

This enumeration specifies the value assigned to pixels that are on the boundary between solid and hole regions in a region tree. This is one of the parameters passed to the **cfRasterize()** function.

Value	Meaning
eUseInterpolatedPelValue	Use a grey level that interpolates the existing grey level and the foreground grey level using the fraction of the pixel area covered by region. This is the anti-aliasing mode.
eUseLargestPortionPelValue	Use the foreground grey level if the region covers at least half of the boundary pixel. Otherwise, do not change the pel value.

Value	Meaning
eUseForegroundPelValue	Use the foreground grey level for boundary pixels.
eUseBackgroundPelValue	Do not change boundary pixel values. The existing pixel values remain.
kDefaultBoundaryFillMode	The default boundary fill mode is *eUseInterpolatedPelValue.*

Static Functions

RasterizeSampParams

```
static const
  ccShape::ccSampleParams &RasterizeSampParams();
```

Specifies the sampling parameters used to sample region trees that will be rasterized to a pel buffer using the **cfRasterize()** global function. Returns a reference to a **ccShape::ccSampleParams** object with the following parameter values.

Parameter	Value
Tolerance	0.1
Spacing	0.0
Maximum number of points	*ckMaxInt32*
Compute tangents	False
Duplicate corners	False

ccRect

```
#include <ch_cvl/shapes.h>

class ccRect : public ccShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a rectangle oriented to the client coordinate system's x- and y-axes. You specify the location of the upper left corner of the rectangle and its size. For a general rectangle whose sides are not parallel to the axes, see **ccGenRect**.

Constructors/Destructors

ccRect

```
ccRect();

ccRect(const cc2Vect& ul, const cc2Vect& sz);

ccRect(const ccPelRect &rect);
```

- `ccRect();`

  Default constructor. Constructs a degenerate rectangle whose upper left corner is at (0,0) and whose size is (0,0).

- `ccRect(const cc2Vect& ul, const cc2Vect& sz);`

  Constructs a rectangle at the specified location and of the specified size.

  Parameters

ul	The location of the upper left corner of the rectangle.
sz	The size of the rectangle as a 2-element vector specifying the rectangle's width and height.

  Notes

  Both components of *sz* must be non-negative.

- `ccRect(const ccPelRect &rect);`

  Constructs a rectangle from the supplied **ccPelRect**.

  Parameters

  rect The **ccPelRect** from which to construct this object.

  Notes

  The origin and size are set to those of *rect*.

Operators

operator== `bool operator==(const ccRect& other) const;`

Returns true if this rectangle is equal to another rectangle.

Parameters

other The other rectangle.

operator!= `bool operator!=(const ccRect& other) const;`

Returns true if this rectangle is not equal to another rectangle.

Parameters

other The other rectangle.

operator& `ccRect operator&(const ccRect& r) const;`

Returns the intersection of this rectangle and the rectangle *r*.

Parameters

r The other rectangle.

Public Member Functions

ul `const ccPoint& ul() const;`

Returns the coordinates of the corner with the minimum x- and y- values in client coordinates. This is the upper-left corner of a rectangle in left-handed, non-rotated client coordinates.

ur

```
ccPoint ur() const;
```

Returns the coordinates of the corner with the maximum x- and minimum y- values in client coordinates. This is the upper-right corner of a rectangle in left-handed, non-rotated client coordinates.

lr

```
ccPoint lr() const;
```

Returns the coordinates of the corner with the maximum x- and y- values in client coordinates. This is the lower-right corner of a rectangle in left-handed, non-rotated client coordinates.

ll

```
ccPoint ll() const;
```

Returns the coordinates of the corner with the minimum x- and maximum y- values in client coordinates. This is the lower-left corner of a rectangle in left-handed, non-rotated client coordinates.

sz

```
const cc2Vect& sz () const;
```

Returns the size of the rectangle as a 2-element vector (width, height).

lSeg

```
ccLineSeg lSeg() const;
```

Returns the left line segment of the rectangle.

rSeg

```
ccLineSeg rSeg() const;
```

Returns the right line segment of the rectangle.

tSeg

```
ccLineSeg tSeg() const;
```

Returns the top line segment of the rectangle.

bSeg

```
ccLineSeg bSeg() const;
```

Returns the bottom line segment of the rectangle.

map

```
ccGenRect map(const cc2Xform& c) const;
```

Returns a generalized rectangle that is the result of mapping this rectangle with the transformation object *c*.

Parameters

c The transformation object.

Notes

Both dimensions of this rectangle must be strictly positive, and the transform must be non-singular. If this is not the case, use **mapshape()** instead of this function.

enclose

```
ccRect enclose(const ccRect& r) const;
```

Returns the smallest rectangle that encloses this rectangle and the rectangle *r*.

Parameters

r The rectangle.

degen

```
bool degen() const;
```

Returns true if this rectangle is *degenerate*. A rectangle is degenerate if both of its dimensions are non-positive (less than or equal to zero).

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this rectangle.

isOpenContour

```
virtual bool isOpenContour() const;
```

Returns true if this shape is an open contour. For rectangles, this function always returns false. See **ccShape::isOpenContour()** for more information.

isRegion

```
virtual bool isRegion() const;
```

Returns true if this shape is a region. For rectangles, this function always returns true, even for rectangles that are degenerate. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For rectangles, this function always returns true. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

Returns true if the set of points that lie on the boundary of this shape is empty. For rectangles, this function always returns false. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

This function returns true for most rectangles. It returns false if all four vertices of the rectangle are coincident, in which case the rectangle collapses to a single point and there is no tangent. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For rectangles, this function always returns false. See **ccShape::isDecomposed()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For rectangles, this function always returns false. See **ccShape::reverse()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

For rectangles, this function always returns true. Rectangles are always right-handed. See **ccShape::isRightHanded()** for more information.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this rectangle. See **ccShape::boundingBox()** for more information.

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point on the boundary of this rectangle to the given point. If the nearest point is not unique, one of the nearest points will be returned.

Parameters

p	The point.

See **ccShape::nearestPoint()** for more information.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
  ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this shape.

Parameters

params	Specifies details of how the sampling should be done.

result Result object to which position and tangent chains are appended.

Notes

If **params.computeTangents()** is true, this function ignores rectangles for which **hasTangent()** is false.

See **ccShape::sample()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this rectangle mapped by *X*.

Parameters

X The transformation object.

Notes

If *X* is the identity transform or this rectangle is degenerate, this function returns a **ccRect**. If both of these conditions are false, this function returns a **ccGenRect** if *X* is nonsingular, and a **ccPolyline** if *X* is singular.

See **ccShape::mapShape()** for more information.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a right-handed **ccContourTree** consisting of four connected **ccLineSeg**s. See **ccShape::decompose()** for more information.

within

```
virtual bool within(const cc2Vect& p) const;
```

Returns true if the given point is within this rectangle.

Parameters

p The point.

See **ccShape::within()** for more information.

enclosePelRect

```
ccPelRect enclosePelRect() const;
```

Returns the minimum **ccPelRect** that encloses this **ccRect**.

Deprecated Members

These functions are deprecated and are provided for backwards compatibility only. Use the appropriate functions provided by **ccShape** instead.

encloseRect

```
ccRect encloseRect() const;
```

Use **boundingBox()** instead of this function.

distToPoint

```
double distToPoint(const cc2Vect& v) const;
```

Use **distanceToPoint()** instead of this function.

ccRectangle

```
#include <ch_cvl/rect.h>

template <class T> class ccRectangle;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

The **ccRectangle** class is a template class for describing rectangular regions. Rectangles have an origin, also known as the upper left component, and a height and a width.

T — Template parameter specifying the type of the component values. CVL provides instantiations of **ccRectangle** for the types **c_Int32** and **double**.

Constructors/Destructors

ccRectangle

```
ccRectangle();

ccRectangle(T w, T h);

ccRectangle(T x, T y, T w, T h);

ccRectangle(const ccPair<T>& v1, const ccPair<T>& v2);
```

- `ccRectangle();`

  Makes this rectangle have origin (0,0), with width and height both equal to zero.

- `ccRectangle(T w, T h);`

  Makes this rectangle have width *w* and height *h*. If *w* and *h* are both non-negative, the origin is (0,0).

  Parameters

  w — The width of the rectangle.

  h — The height of the rectangle.

Notes

If *w* is negative, the rectangle's origin x-component is *w* (negative), and its width is *-w* (positive). Similarly, if *h* is negative, the rectangle's origin y-component is *h* (negative), and its height is *-h* (positive).

- `ccRectangle(T x, T y, T w, T h);`

Makes this rectangle have the indicated origin, width, and height.

Parameters

x	The origin's x-component.
y	The origin's y-component.
w	The width of the rectangle.
h	The height of the rectangle.

Notes

If *w* is negative, the rectangle's origin x-component is *x+w*, and its width is *-w* (positive). Similarly, if *h* is negative, the rectangle's origin y-component is *y+h,* and its height is *-h* (positive).

- `ccRectangle(const ccPair<T>& v1, const ccPair<T>& v2);`

Makes this rectangle the minimum enclosing rectangle of the indicated two points.

Parameters

v1	The origin (upper left component)
v2	The lower right component

Operators

operator== `bool operator== (const ccRectangle<T>& other) const;`

Returns true if this rectangle and *other* have the same origin, width, and height.

Parameters

other	The other rectangle to test.

operator!= `bool operator!= (const ccRectangle<T>& other) const;`

Returns true if this rectangle and *other* do not have the same origin, width, and height.

Parameters

other	The other rectangle to test.

operator&

```
ccRectangle<T> operator& (const ccRectangle<T>& other)
  const;
```

Return the rectangle that is the intersection of this rectangle and *other*. This operator has the same effect as calling:

```
ccRectangle<T>::intersect(other, false);
```

Parameters

other The other rectangle to test against this one

Notes

To compute the Greatest Common Rectangle (GCR) of several rectangles, use the following:

```
gcr = (rect1 & rect2 & rect3 & rect4 & ...);
if (!gcr.isNull())
  ...
```

operator&=

```
ccRectangle<T>& operator&= (const ccRectangle<T>& other);
```

Modifies this rectangle to be the intersection of this rectangle and *other*. Returns a reference to this modified object.

Parameters

other The other rectangle to test against this one.

operator|

```
ccRectangle<T> operator| (const ccRectangle<T>& other)
  const;
```

Returns the smallest rectangle that completely encloses this rectangle and *other*. If this rectangle is a null rectangle or if *other* is a null rectangle, this function returns the non-null rectangle. If both this rectangle and other are null rectangles, this function returns a null rectangle whose origin is undefined.

This operator has the same effect as calling:

```
ccRectangle<T>::enclose(other);
```

Parameters

other The other rectangle to consider when constructing the smallest enclosing rectangle.

Notes

To compute the minimum enclosing rectangle of several rectangles, use the following:

```
minRect = (rect1 | rect2 | rect3 | rect4 | ...);
```

operator|=

```
ccRectangle<T>& operator|= (const ccRectangle<T>& other);
```

Modifies this rectangle to be the minimum enclosing rectangle of this rectangle and *other*.

Parameters

other	The other rectangle to consider when constructing the smallest enclosing rectangle.

Public Member Functions

origin

```
const ccPair<T>& origin() const;
void origin(const ccPair<T>&) const;
```

- `const ccPair<T>& origin() const;`

  Returns the origin, or upper left corner, of this rectangle.

- `void origin(const ccPair<T>& orig) const;`

  Sets the origin of this rectangle without modifying its width or height.

  Parameters

orig	The new origin of the rectangle.

translate

```
void translate(const ccPair<T>& trans);
```

Moves the origin of this rectangle without modifying its width or height, by adding *trans* to the rectangle's origin.

Parameters

trans	The point to add to the rectangle's origin

ul

```
const ccPair<T>& ul() const;
```

Returns the coordinates of the corner with the minimum x- and y- values in client coordinates. Same as **origin()**. This is the upper-left corner of a rectangle in left-handed, non-rotated client coordinates.

ur

```
ccPair<T> ur() const;
```

Returns the coordinates of the corner with the maximum x- and minimum y- values in client coordinates. This is the upper-right corner of a rectangle in left-handed, non-rotated client coordinates.

ll

```
ccPair<T> ll() const;
```

Returns the coordinates of the corner with the minimum x- and maximum y- values in client coordinates. This is the lower-left corner of a rectangle in left-handed, non-rotated client coordinates.

lr

```
const ccPair<T>& lr() const;
```

Returns the coordinates of the corner with the maximum x- and y- values in client coordinates. This is the lower-right corner of a rectangle in left-handed, non-rotated client coordinates.

width

```
T width() const;
void width(T w);
```

- `T width() const;`

  Returns the width of this rectangle.

- `void width(T w);`

  Sets the width of this rectangle without changing its origin.

  Parameters

  w The new width of the rectangle.

  Notes

  If *w* is negative, its value is added to the x component of the rectangle's origin, and the rectangle's width is set to *-w* (positive).

height

```
T height() const;
void height(T h);
```

- `T height() const;`

  Returns the height of this rectangle.

- `void height(T h);`

Sets the height of this rectangle without changing its origin.

Parameters

h The new height of the rectangle.

Notes

If *h* is negative, its value is added to the y component of the rectangle's origin, and the rectangle's height is set to *-h* (positive).

---

size

```
ccPair<T> size() const;
void size(const ccPair<T>&);
```

- `ccPair<T> size() const;`

Returns the size of this rectangle. The width is the x component of the returned pair; the height is the y component.

- `void size(const ccPair<T>& siz);`

Sets the width from the x component of *siz*, and sets the height from the y component of *siz*.

Parameters

siz The new size of the rectangle.

Notes

See **width()** and **height()** on page 2703 for a description of how this function handles negative size values.

isNull

```
bool isNull(void) const;
```

Returns true if this rectangle is a null rectangle, that is, its height is zero or its width is zero.

overlaps

```
bool overlaps(const ccRectangle<T>& other,
  bool touching=false) const;
```

Returns true if the *other* overlaps this rectangle.

Parameters

other The rectangle to compare to this rectangle.

touching If true, rectangles that touch at their borders are considered overlapping. If not true, rectangles are considered to overlap only if a portion of one rectangle's border is within the other rectangle

Notes

A rectangle, *rectA*, overlaps a null rectangle, *rectNull*, only if *touching* is true and the following expression is true:

```
rectA.contains(rectNull)
```

contains

```
bool contains(const ccRectangle<T>& rect) const;

bool contains(const ccPair<T>& pt) const;
```

- `bool contains(const ccRectangle<T>& rect) const;`

Returns true if the indicated rectangle is contained within this rectangle.

Parameters

rect The rectangle to test for inclusion in this rectangle.

- `bool contains(const ccPair<T>& pt) const;`

Returns true if the indicated point is contained within this rectangle.

Parameters

pt The point to test for inclusion in this rectangle.

Notes

A point that barely touches the bottom or right borders of a rectangle is not contained in the rectangle.

intersect

```
ccRectangle<T> intersect(const ccRectangle<T>& other,
  bool throwIfNoOverlap=false) const;
```

Returns a rectangle that is the intersection of *other* and this rectangle. If *throwIfNoOverlap* is true, this function throws an error if the rectangles do not overlap. Otherwise, the function returns a null rectangle. It is your responsibility not to use this null rectangle for further rectangle operations.

Parameters

other The other rectangle to test against this one

throwIfNoOverlap
If true, throw an error if the rectangles don't overlap.

Throws

ccRect::NoOverlap
The two rectangles do not overlap and *throwIfNoOverlap* is set.

Notes
Rectangles that just barely touch produce a null ccRectangle whose origin is valid.

enclose

```
ccRectangle<T> enclose(const ccRectangle<T>& other) const;
```

Returns the smallest rectangle that completely encloses this rectangle and *other*. If this rectangle is a null rectangle or if *other* is a null rectangle, this function returns the non-null rectangle. If both this rectangle and other are null rectangles, this function returns a null rectangle whose origin is undefined.

Parameters

other — The other rectangle to consider when constructing the smallest enclosing rectangle.

trim

```
ccRectangle<T> trim(T left, T right, T top, T bottom,
  bool throwIfNegativeSize=false) const;
```

Returns a copy of this rectangle whose edges are trimmed by the amounts specified in *left*, *right*, *top*, and *bottom*. Positive values trim, or shrink, the rectangle. Negative values grow the rectangle.

If *left + right* > **width()** or if *top + bottom* > **height()** and *throwIfNegativeSize* is false, the resulting values are treated as negative values passed to **width()** and **height()**.

Parameters

left — The amount by which to shrink (positive values) or grow (negative values) the left edge of the rectangle

right — The amount by which to shrink (positive values) or grow (negative values) the right edge of the rectangle

top — The amount by which to shrink (positive values) or grow (negative values) the top edge of the rectangle

bottom — The amount by which to shrink (positive values) or grow (negative values) the bottom edge of the rectangle

throwIfNegativeSize
If true, throw an error if *left + right* > **width()** or if *top + bottom* > **height()**.

Throws

ccRect::NegativeSize
: The trimming requested results in negative size values for rectangle and *throwIfNegativeSize* is set.

transpose

```
ccRectangle<T> transpose () const;
```

Returns the transposition of this object. The transposition operation for two rectangles A and B such that

```
A = B.transpose();
```

is defined as follows:

```
A.origin().x() = B.origin.y();
A.origin().y() = B.origin.x();
A.height()     = B.width();
A.width()      = B.height();
```

ccRectangle

ccRegionTree

```
#include <ch_cvl/shaptree.h>

class ccRegionTree : public ccShapeTree;
```

Class Properties

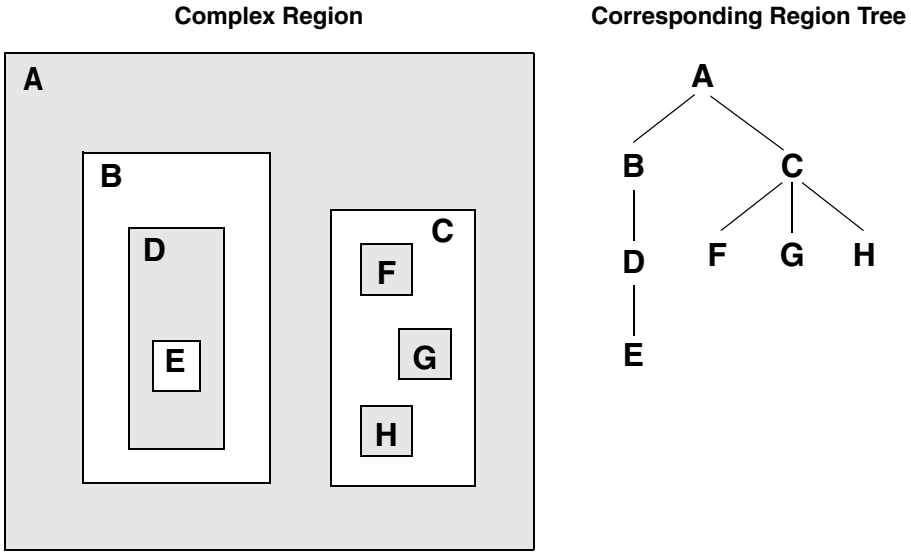
Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class is a concrete class for maintaining hierarchies that define regions of the plane. The regions may or may not be connected and may have holes. Holes may contain solid regions, which may in turn contain more holes, ad infinitum.

The only allowable children for a **ccRegionTree** are other **ccRegionTree**s. In addition to children, all **ccRegionTree**s have a boundary. The boundary must be any shape, other than a **ccRegionTree**, for which **isRegion()** is true. Possibilities include primitive shapes, such as **ccEllipse**, or closed **ccContourTree**s. Note that **ccRegionTree**s have a different structure than other hierarchical shapes. Primitive shapes are stored at the internal nodes of the hierarchy as boundaries, not as leaf nodes in the hierarchy.

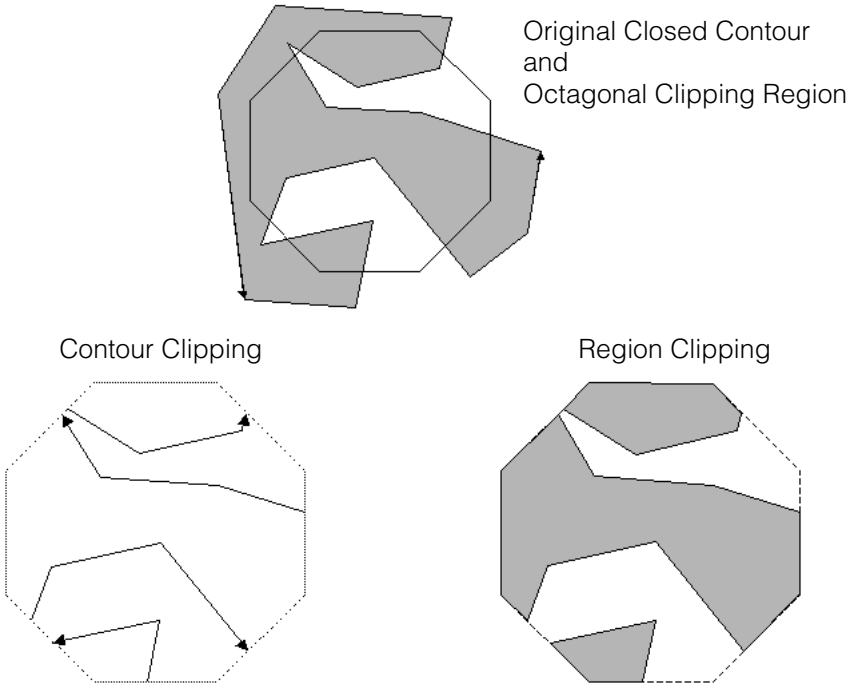
If node A is a parent of node B in a **ccRegionTree**, then B is a hole within solid region A, or B is a solid region within hole A. Thus, the boundary of the root of a **ccRegionTree** encloses the boundaries of all of its descendants. The following figure shows a complex

region and the corresponding region tree. Solid regions are grey and holes are white. In this example the boundaries are all rectangles, but in general they can be any region shapes other than a **ccRegionTree**s.



Multiple disjoint solid regions can be represented in a single **ccRegionTree** by enclosing all of them with a hole region forming the root of the hierarchy. Such *phantom holes* are not rasterized. They are also excluded from certain geometric calculations, such as **area()** and **perimeter()**.

ccRegionTrees are clipped as regions rather than contours. The following figure illustrates the difference between region clipping and contour clipping.



Contour clipping produces a set of contours inside the clipping region. Information regarding whether the contours came from the same continuous boundary and their ordering along such a boundary is lost. Contour clipping is appropriate for applications such as PatMax training where the contours, not the area enclosed by them, is significant.

Region clipping produces a set of regions, bounded by closed contours, inside the clipping region. It does this by appropriately joining the contours produced by contour clipping and adding portions of the clipping region boundary when necessary. Region clipping is appropriate for applications like such as rasterization where the area enclosed by the contours is significant.

Constructors/Destructors

ccRegionTree

```
ccRegionTree();

explicit ccRegionTree(const ccShape &boundary,
  bool hole = false);

explicit ccRegionTree(const ccShapePtrh_const &boundary,
  bool hole = false);

ccRegionTree(const ccRegionTree &orig);
```

- `ccRegionTree();`

  Default constructor. Constructs a solid (non-hole) **ccRegionTree** with a boundary that is a degenerate circle (zero radius, centered at the origin) and no children.

  Notes

  A default constructed **ccRegionTree** is generally not useful. This method is provided to support STL vectors and the complex persistence framework, which both require a default constructor.

- `explicit ccRegionTree(const ccShape &boundary, bool hole = false);`

  Constructs a **ccRegionTree** with the given boundary and no children. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

  boundary — The boundary of the region tree, specified as a reference to a **ccShape**.

  hole — If true, the region enclosed by the boundary is set to a hole region; if false (the default), it is set to a solid region.

  Notes

  This version of the constructor copies the boundary, specified as a reference to a **ccShape**, into this region tree using **clone()**. This version is somewhat slower than the pointer handle version (see the next overload). However, Cognex recommends using this version unless there is a compelling reason not to.

  Throws

  ccShapesError::NotRegion
  boundary is not a region shape.

  ccShapesError::BadGeom
  boundary is neither a primitive shape nor a **ccContourTree**.

- ```
  explicit ccRegionTree(const ccShapePtrh_const &boundary,
    bool hole = false);
  ```

  Constructs a **ccRegionTree** with the given boundary and no children. This constructor is used for explicit construction only and not for implicit conversions.

  Parameters

  boundary The boundary of the region tree, specified as a pointer handle to a **ccShape**.

  hole If true, the region enclosed by the boundary is set to a hole region; if false (the default), it is set to a solid region.

  Notes

  This version of the constructor inserts the boundary directly into the region tree and is, therefore, faster than the reference version (see the previous overload). In this version, this region tree takes ownership of the object to which *boundary* points, and you should not modify the object after this constructor has been invoked. This version of the constructor is provided as an optimization. Cognex recommends using the reference version unless there is a compelling reason not to.

  Throws

  ccShapesError::NotRegion
  boundary is not a region shape.

  ccShapesError::BadGeom
  boundary is neither a primitive shape nor a **ccContourTree**.

- ```
  ccRegionTree(const ccRegionTree &orig);
  ```

  Copy constructor. Constructs a **ccRegionTree** that is a copy of the supplied **ccRegionTree**.

  Parameters

  orig The original region tree.

  Notes

  The newly constructed **ccRegionTree** is identical to *orig* in all respects, except that it is always the root of a hierarchy. Hence, **isRoot()** always returns true for the newly constructed **ccRegionTree**, regardless of the value of **orig.isRoot()**.

  Copies and assignments are shallow, that is, they are achieved by copying pointer handles. Hence, both copy construction and assignment lead to sharing of children between different **ccShapeTree**s.

Operators

operator==

```
bool operator==(const ccRegionTree &rhs) const;
```

Returns true if and only if this **ccRegionTree** is equal to *rhs*. Two **ccRegionTree**s are equal if all of the following conditions are true:

- They are equal as **ccShapeTree**s.
- Their corresponding boundaries are all equal and have the same solid/hole status.
- They are either both roots of a hierarchy, or both not roots.

Parameters

rhs The other **ccRegionTree**.

operator!=

```
bool operator!=(const ccRegionTree &rhs) const;
```

Returns true if and only if this **ccRegionTree** is not equal to *rhs*.

Parameters

rhs The other **ccRegionTree**.

Public Member Functions

isHole

```
bool isHole() const;
```

Returns true if the root boundary of this **ccRegionTree** encloses a hole region, false if it encloses a solid region.

isRoot

```
bool isRoot() const;
```

Returns true if and only if this **ccRegionTree** is the root of a hierarchy (that is, has no parent).

Notes

This method is primarily useful for detecting phantom holes, which are holes at the root of a hierarchy used to group disjoint solid regions into a single **ccRegionTree**. Phantom holes are not rasterized. They should also be excluded from certain geometric calculations, such as **area()** and **perimeter()**.

boundary

```
void boundary(const ccShape &shape);
void boundary(const ccShapePtrh_const &shape);
const ccShape &boundary() const;
```

- `void boundary(const ccShape &shape);`

Sets the boundary of this **ccRegionTree**. The hole status of the boundary remains unchanged.

Parameters

shape The shape that defines the boundary of this region tree.

Notes

This version of **boundary()** copies the new boundary into this **ccRegionTree** indirectly using **clone()**. It is somewhat slower than the pointer handle version (see next overload), which inserts the boundary directly. However, Cognex recommends using this version rather than the pointer handle version unless there is a compelling reason not to.

Throws

ccShapesError::NotRegion
boundary is not a region shape.

ccShapesError::BadGeom
boundary is a **ccRegionTree**.

- `void boundary(const ccShapePtrh_const &shape);`

Sets the boundary of this **ccRegionTree**. The hole status of the boundary remains unchanged.

Parameters

shape The shape that defines the boundary of this region tree.

Notes

A valid boundary shape is any shape for which **isRegion()** is true other than another **ccRegionTree**, such as a closed primitive shape (for example, **ccCircle**) or a closed **ccContourTree**.

This version of **boundary()** inserts the new boundary directly into this **ccRegionTree**. It is faster than the reference version (see previous overload), which copies the boundary indirectly using **clone()**. In this version, this **ccRegionTree** takes ownership of the object to which *boundary* points, and you should not modify the object after this constructor has been invoked. For this reason, Cognex recommends using the reference version unless there is a compelling reason not to.

Throws

ccShapesError::NotRegion
boundary is not a region shape.

ccShapesError::BadGeom
boundary is a **ccRegionTree**.

- `const ccShape &boundary() const;`

Returns the root boundary of this **ccRegionTree**. The boundary is a shape for which **isRegion()** is true. It may be either a primitive shape or a **ccContourTree**.

flip

```
ccRegionTreePtrh flip() const;
```

Returns a new **ccRegionTree** that is identical to this one, except that it has the opposite solid/hole status. Each of the descendants of returned tree also has the opposite solid/hole status from the corresponding descendant in this **ccRegionTree**.

insertChild

```
virtual void insertChild(c_Int32 idx,
 const ccShape &child);

virtual void insertChild(c_Int32 idx,
 const ccShapePtrh_const &child, bool direct = false);
```

- `virtual void insertChild(c_Int32 idx,`
  `const ccShape &child);`

Inserts the given child into this **ccRegionTree** at the given index.

Parameters

idx The index.

child The child to insert.

Throws

ccShapesError::BadGeom
An attempt was made to insert a child that is not itself a **ccRegionTree**.

An attempt was made to insert a child that has the same solid/hole status as this **ccRegionTree**. The solid/hole status of a

child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of a child, if necessary, prior to invoking this method.

ccShapesError::BadIndex
idx is less than zero or greater than **numChildren()** before the insertion.

See **ccShapeTree::insertChild()** for more information.

- ```
  virtual void insertChild(c_Int32 idx,
    const ccShapePtrh_const &child, bool direct = false);
  ```

Inserts the given child into this **ccRegionTree** at the given index. The child is inserted by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

idx The index.

child The child to insert.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom
An attempt was made to insert a child that is not itself a **ccRegionTree**.

An attempt was made to insert a child that has the same solid/hole status as this **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of a child, if necessary, prior to invoking this method.

ccShapesError::BadIndex
idx is less than zero or greater than **numChildren()** before the insertion.

See **ccShapeTree::insertChild()** for more information.
  ```

insertChildren

```
virtual void insertChildren(c_Int32 idx,
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Inserts the given children into this **ccRegionTree** at the given index. The children are inserted by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

idx The index.

children The vector of children to insert.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom

An attempt was made to insert a child that is not itself a **ccRegionTree**.

An attempt was made to insert children that have the same solid/hole status as this **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of the children, if necessary, prior to invoking this method.

ccShapesError::BadIndex

idx is less than zero or greater than **numChildren()** before the insertion.

See **ccShapeTree::insertChildren()** for more information.

---

addChild

```
virtual void addChild(const ccShape &child);

virtual void addChild(const ccShapePtrh_const &child,
  bool direct = false);
```

- ```
  virtual void addChild(const ccShape &child);
  ```

Appends the given child to the end of this **ccRegionTree**.

Parameters

child The child to add.
  ```

Throws

ccShapesError::BadGeom

An attempt was made to add a child that is not itself a **ccRegionTree**.

An attempt was made to add a child that has the same solid/hole status as this **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of a child, if necessary, prior to invoking this method.

See **ccShapeTree::addChild()** for more information.

- ```
  virtual void addChild(const ccShapePtrh_const &child,
    bool direct = false);
  ```

Appends the given child to the end of this **ccRegionTree**. The child is added by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

child To child to add.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom

An attempt was made to add a child that is not itself a **ccRegionTree**.

An attempt was made to add a child that has the same solid/hole status as this **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of a child, if necessary, prior to invoking this method.

See **ccShapeTree::addChild()** for more information.

addChildren

```
virtual void addChildren(
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Adds the given children to this **ccRegionTree**. The children are added by copy insertion or direct insertion, depending on the status of the *direct* flag.
```

Parameters

children The vector of children to add.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom

An attempt was made to add children that are not themselves **ccRegionTree**s.

An attempt was made to insert children that have the same solid/hole status as this **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of the children, if necessary, prior to invoking this method.

See **ccShapeTree::addChildren()** for more information.

---

replaceChild

```
virtual void replaceChild(c_Int32 idx,
 const ccShape &child);

virtual void replaceChild(c_Int32 idx,
 const ccShapePtrh_const &child, bool direct = false);
```

- ```
  virtual void replaceChild(c_Int32 idx,
   const ccShape &child);
  ```

Replaces the child at the given index with the given child.

Parameters

idx The index.

child The child replacing the indexed child.

Throws

ccShapesError::BadGeom

An attempt was made to replace with a child that is not itself a **ccRegionTree**.

An attempt was made to replace with a child that has the same solid/hole status as this **ccRegionTree**. The solid/hole status of a
  ```

child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of a child, if necessary, prior to invoking this method.

ccShapesError::BadIndex
idx is less than zero or greater than or equal to **numChildren()** before the replacement.

See **ccShapeTree::replaceChild()** for more information.

- ```
  virtual void replaceChild(c_Int32 idx,
    const ccShapePtrh_const &child, bool direct = false);
  ```

Replaces the child of this **ccRegionTree** at the given index with the given child. The child is replaced by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

idx The index.

child The child replacing the indexed child.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom
An attempt was made to replace with a child that is not itself a **ccRegionTree**.

An attempt was made to replace with a child that has the same solid/hole status as this **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse the solid/hole status of a child, if necessary, prior to invoking this method.

ccShapesError::BadIndex
idx is less than zero or greater than or equal to **numChildren()** before the replacement.

See **ccShapeTree::replaceChild()** for more information.
  ```

replaceChildren

```
virtual void replaceChildren(
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Replaces all of the children of this **ccRegionTree** with the given children. The children are replaced by copy insertion or direct insertion, depending on the status of the *direct* flag.

Parameters

children The vector of children replacing the current children.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it.

Throws

ccShapesError::BadGeom

An attempt was made to replace the current children with children that are not themselves **ccRegionTree**s.

An attempt was made to replace with children that have the same solid/hole status as their parent **ccRegionTree**. The solid/hole status of a child must be the opposite of its parent. You can use **flip()** to reverse a child's solid/hole status, if necessary, prior to invoking this method.

See **ccShapeTree::replaceChildren()** for more information.

clone

```
virtual ccShapePtrh clone() const;
```

Returns a pointer to a copy of this region tree.

isRegion

```
virtual bool isRegion() const;
```

For **ccRegionTree**s, this method always returns true. See **ccShape::isRegion()** for more information.

isFinite

```
virtual bool isFinite() const;
```

For **ccRegionTree**s, this method always returns true. Region shapes are necessarily finite. See **ccShape::isFinite()** for more information.

isEmpty

```
virtual bool isEmpty() const;
```

For **ccRegionTree**s, this method always returns false. All boundaries in a region tree hierarchy must be region shapes, which are always non-empty. See **ccShape::isEmpty()** for more information.

hasTangent

```
virtual bool hasTangent() const;
```

Returns true if any of the boundaries in the hierarchy has a tangent. See **ccShape::hasTangent()** for more information.

isDecomposed

```
virtual bool isDecomposed() const;
```

For **ccRegionTree**s, this method always returns false. See **ccShape::isDecomposed()** for more information.

isOpenContour

```
virtual bool isOpenContour() const;
```

For **ccRegionTree**s, this method always returns false. See **ccShape::isOpenContour()** for more information.

isRightHanded

```
virtual bool isRightHanded() const;
```

Returns true if this region tree is right handed.

Notes

A **ccRegionTree** is right handed if and only if its root boundary encloses a solid region. Thus, **isRightHanded()** is equivalent to **!isHole()** for **ccRegionTree**s. Sampling a **ccRegionTree** produces right-handed boundaries for solid regions and left-handed boundaries for hole regions.

See **ccShape::isRightHanded()** for more information.

isReversible

```
virtual bool isReversible() const;
```

For **ccRegionTree**s, this function always returns false. See **ccShape::isReversible()** for more information.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if the boundary of the hierarchy that most tightly encloses the given point corresponds to a solid region. Returns false if the boundary of the root node of the hierarchy does not enclose the given point.

Parameters

p The point.

See **ccShape::within()** for more information./

nearestPoint

```
virtual cc2Vect nearestPoint(const cc2Vect &p) const;
```

Returns the nearest point along the boundary of this shape to the given point.

Parameters

p The point.

Notes

The nearest point to *p* is taken among all boundaries in the hierarchy, excluding phantom holes.

sample

```
virtual void sample(const ccShape::ccSampleParams &params,
 ccShape::ccSampleResult &result) const;
```

Returns sample positions, and possibly tangents, along this region tree.

Parameters

params Parameters object specifying details of how the sampling should be done.

result Result object to which position and tangent chains are stored.

Notes

A separate chain is generated for each boundary in the hierarchy. Each chain is an approximating polygon to a boundary. The polygon is right-handed for boundaries of solid regions and left-handed for boundaries of hole regions. This is true even if, for example, a right-handed polygonal hole is added to a **ccRegionTree**. The **sample()** method reverses the sample points when necessary. The reason for this is that making solid boundaries and hole boundaries have opposite handedness is the best approach for algorithms for which handedness matters. For example, algorithms that rasterize or compute areas of polygonal solid regions can easily be extended to handle regions with holes if the holes are guaranteed to have the opposite handedness to the solid regions.

boundingBox

```
virtual ccRect boundingBox() const;
```

Returns the smallest rectangle that encloses this region tree.

Notes

The enclosing rectangle of this **ccRegionTree** is the smallest rectangle that encloses all the component primitives of the boundaries. This class must override the **boundingBox()** method of **ccShapeTree** because the primitives are stored at internal nodes of the hierarchy instead of at the leaf nodes.

See **ccShape::boundingBox()** for more information.

perimeter

```
virtual double perimeter() const;
```

Returns the perimeter of this region tree.

Notes

Phantom boundaries are not included in the perimeter.

See **ccShape::perimeter()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this region tree to the given point, as determined by **nearestPoint()**.

Parameters

info	Shape information for this region tree.
point	The point.

See **ccShape::nearestPerimPos()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this region tree mapped by the transformation object *X*.

Parameters

X	The transformation object.

Notes

This method generates a new **ccRegionTree** obtained by mapping each of its boundaries by *X*. The new boundaries are hierarchical refinements of the original boundaries.

Mapping by an affine transform does not change the topology of the region tree; enclosure relationships are maintained. Therefore, the structure of the hierarchy is preserved.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a **ccGeneralShapeTree** that is the decomposed version of the equivalent **ccGeneralShapeTree**. See **ccShape::decompose()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this region tree over the given perimeter range.

Parameters

info	Shape information for this region tree.
range	The perimeter range.

See **ccShape::subShape()** for more information.

ccRepBase

```
#include <ch_cvl/handle.h>

class ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

ccRepBase allows any class to be accessed through a pointer handle. For more information on pointer handles, see the section *CVL Programming Conventions* in the chapter *CVL Programming Overview* in the *CVL User's Guide*.

To make a class **ccFoo** accessible through a pointer handle, define **ccFoo** so that it is derived from **ccRepBase**:

```
class ccFoo : public virtual ccRepBase {
  ...
};
```

Use **ccPtrHandle** on page 2659 and **ccPtrHandle_const** on page 2663 to define pointer handles.

Constructors/Destructors

This class is never used on its own. It is always used as a base class.

Public Member Functions

refc

```
c_Int32 refc() const;
```

Returns the number of pointer handles that refer to this object.

ccRepBase

ccRGB

```
#include <ch_cvl/color.h>

class ccRGB;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class represents an RGB color, a color composed of red, green, and blue components. Most CVL functions that operate on colors use **ccColor** on page 1021, a class related to this class. **ccColor** provides stock colors defined as constants and as functions.

Constructors/Destructors

ccRGB

```
ccRGB(c_UInt8 r = 0, c_UInt8 g = 0, c_UInt8 b = 0);
```

Creates a new RGB color.

Parameters

r	The red component. Must be a value from 0 to 255.
g	The green component. Must be a value from 0 to 255.
b	The blue component. Must be a value from 0 to 255.

Operators

operator!=

```
bool operator!= (const ccRGB& c);
```

Returns true if this color is not equal to another color.

Parameters

c	The color to compare to this color.

operator==

```
bool operator== (const ccRGB& c) const;
```

Returns true if this color is equal to another color.

Parameters

c The color to compare to this color.

operator<

```
bool operator< (const ccRGB& c) const;
```

Returns true if the intensity of this color is less than the intensity of another color. The intensity of a color is the average of all three components.

Parameters

c The color compare to this color.

operator>

```
bool operator> (const ccRGB& c) const;
```

Returns true if the intensity of this color is greater than the intensity of another color. The intensity of a color is the average of all three components.

Parameters

c The color compare to this color.

Public Member Functions

r

```
c_UInt8 r() const;
void r(c_UInt8 c);
```

- `c_UInt8 r() const;`

  Returns this color's red component.

- `void r(c_UInt8 c);`

  Sets this color's red component.

  Parameters

  c The red component. Must be a value from 0 to 255.

g

```
c_UInt8 g() const;
void g(c_UInt8 c);
```

- `c_UInt8 g() const;`

  Returns this color's green component.

- `void g(c_UInt8 c);`

  Sets this color's green component.

  Parameters

c	The green component. Must be a value from 0 to 255.

b

```
c_UInt8 b() const;
void b(c_UInt8 c);
```

- `c_UInt8 b() const;`

  Returns this color's blue component.

- `void b(c_UInt8 c);`

  Sets this color's blue component.

  Parameters

c	The blue component. Must be a value from 0 to 255.

rgb

```
void rgb(c_UInt8 r, c_UInt8 g, c_UInt8 b);
c_UInt32 rgb() const;
```

- `void rgb(c_UInt8 r, c_UInt8 g, c_UInt8 b);`

  Sets this color's RGB values.

  Parameters

r	The red component. Must be a value from 0 to 255.
g	The green component. Must be a value from 0 to 255.
b	The blue component. Must be a value from 0 to 255.

- `c_UInt32 rgb() const;`

  Returns this color's RGB value as a single 32-bit value in the format `0x00rrggbb`.

bgr

`c_UInt32 bgr() const;`

Returns this color's RGB value as a single 32-bit value in the format `0x00bbggrr`.

rgb15 `c_UInt16 rgb15() const;`

Returns this color's RGB value as a single "5-5-5" 16-bit value, using 5 bits for each component. The low-order 3 bits of each original component are ignored.

rgb16 `c_UInt16 rgb16() const;`

Returns this color's RGB values as a single "5-6-5" 16-bit value, using 5 bits for the red component, 6 bits for this green component, and 5 bits for the blue component. The low-order bits of each original component are ignored.

ccRGB16AcqFifo

```
#include <ch_cvl/acq.h>

class ccRGB16AcqFifo : public ccAcqFifo;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This class describes acquisition FIFO queues that return 16-bit RGB images.

Notes

There is no need for you to create objects of this type in your program. When you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()** it returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

Constructors/Destructors

ccRGB16AcqFifo

```
ccRGB16AcqFifo(const ccVideoFormat& vf,
  ccFrameGrabber& fg);
```

Creates an idle acquisition FIFO suitable for capturing packed 16-bit color images of the format *vf* from the frame grabber *fg*.

Parameters

vf	The video format of the images to be acquired.
fg	The frame grabber to use to acquire images.

Public Member Functions

baseComplete

```
PelBuffer* baseComplete (ccAcqFailure* result = 0,
  c_UInt32* appTag = 0, bool makeLocal = true,
  double maxWait = HUGE_VAL, bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Returns a pointer to the image of the oldest outstanding acquisition and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one. Returns null if the acquisition failed or if the *maxWait* period elapsed.

Parameters

result — If not null, **result* is set to a description of why the acquisition failed. See **ccAcqFailure** on page 209.

appTag — If not null, **appTag* is set to the value that was passed to **start()**. If the acquisition failed because of too many outstanding acquisitions, or because acquisition was incomplete below then the returned **appTag* value is undefined. For trigger models that do not use **start()** to initiate the acquisition, this value is undefined.

makeLocal — If true, **baseComplete()** forces the image to be local (see **ccPelRoot::move()**). Otherwise, the image remains where it was acquired. For example, if your frame grabber (**ccFrameGrabber**) is also an accelerator (**ccAccelerator**), you might want to set *makeLocal* to false to leave the image on the accelerator to be processed there.

maxWait — The maximum number of seconds to wait for a complete acquisition to become available. The special value *HUGE_VAL* means to wait indefinitely. If the *maxWait* period elapses, **result->isIncomplete()** returns true.

autoStart — If *autoStart* is true and the selected trigger model associated with this FIFO allows **start()** to be invoked, then **start()** is invoked automatically when an acquisition completes. In this case, **start()** is invoked with an *appTag* of zero. If the acquisition is incomplete (**result->isIncomplete()** returns true) or if this function throws, **start()** is not invoked automatically.

startReqStatus — If not NULL, returns whether the acquisition request was processed completely or not. See **ceStartReqStatus** on page 218.

Throws

ccPel::BadWindow

You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.

complete

```
PelBuffer complete (ccAcqFailure* result = 0,
  c_UInt32* appTag = 0, bool makeLocal = true,
  double maxWait = HUGE_VAL, bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Returns the image of the oldest outstanding acquisition and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one. Returns null if the acquisition failed.

Parameters

result — If not null, **result* is set to a description of why the acquisition failed. See **ccAcqFailure** on page 209.

appTag — If not null, **appTag* is set to the value that was passed to **start()**. If the acquisition failed because of too many outstanding acquisitions, or because acquisition was incomplete below then the returned **appTag* value is undefined. For trigger models that do not use **start()** to initiate the acquisition, this value is undefined.

makeLocal — If true, **baseComplete()** forces the image to be local (see **ccPelRoot::move()**). Otherwise, the image remains where it was acquired. For example, if your frame grabber (**ccFrameGrabber**) is also an accelerator (**ccAccelerator**), you might want to set *makeLocal* to false to leave the image on the accelerator to be processed there.

maxWait — The maximum number of seconds to wait for a complete acquisition to become available. The special value *HUGE_VAL* means to wait indefinitely. If the *maxWait* period elapses, **result->isIncomplete()** returns true.

autoStart — If *autoStart* is true and the selected trigger model associated with this FIFO allows **start()** to be invoked, then **start()** is invoked automatically when an acquisition completes. In this case, **start()** is invoked with an *appTag* of zero. If the acquisition is incomplete (**result->isIncomplete()** returns true) or if this function throws, **start()** is not invoked automatically.

startReqStatus If not NULL, returns whether the acquisition request was processed completely or not. See **ceStartReqStatus** on page 218.

Throws

ccPel::BadWindow

You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.

Typedefs

PelBuffer `typedef ccPelBuffer<ccPackedRGB16Pel> PelBuffer;`

ccRGB32AcqFifo

```
#include <ch_cvl/acq.h>

class ccRGB32AcqFifo : public ccAcqFifo;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This class describes acquisition FIFO queues that return 32-bit RGB images.

Notes

There is no need for you to create objects of this type in your program. When you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()** it returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

Constructors/Destructors

ccRGB32AcqFifo

```
ccRGB32AcqFifo(const ccVideoFormat& vf,
  ccFrameGrabber& fg);
```

Creates an idle acquisition FIFO suitable for capturing packed 32-bit color images of the format *vf* from the frame grabber *fg*.

Parameters

vf	The video format of the images to be acquired.
fg	The frame grabber to use to acquire images.

Public Member Functions

baseComplete

```
PelBuffer* baseComplete (ccAcqFailure* result = 0,
  c_UInt32* appTag = 0, bool makeLocal = true,
  double maxWait = HUGE_VAL, bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Returns a pointer to the image of the oldest outstanding acquisition and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one. Returns null if the acquisition failed or if the *maxWait* period elapsed.

Parameters

result — If not null, **result* is set to a description of why the acquisition failed. See **ccAcqFailure** on page 209.

appTag — If not null, **appTag* is set to the value that was passed to **start()**. If the acquisition failed because of too many outstanding acquisitions, or because acquisition was incomplete below then the returned **appTag* value is undefined. For trigger models that do not use **start()** to initiate the acquisition, this value is undefined.

makeLocal — If true, **baseComplete()** forces the image to be local (see **ccPelRoot::move()**). Otherwise, the image remains where it was acquired. For example, if your frame grabber (**ccFrameGrabber**) is also an accelerator (**ccAccelerator**), you might want to set *makeLocal* to false to leave the image on the accelerator to be processed there.

maxWait — The maximum number of seconds to wait for a complete acquisition to become available. The special value *HUGE_VAL* means to wait indefinitely. If the *maxWait* period elapses, **result->isIncomplete()** returns true.

autoStart — If *autoStart* is true and the selected trigger model associated with this FIFO allows **start()** to be invoked, then **start()** is invoked automatically when an acquisition completes. In this case, **start()** is invoked with an *appTag* of zero. If the acquisition is incomplete (**result->isIncomplete()** returns true) or if this function throws, **start()** is not invoked automatically.

startReqStatus — If not NULL, returns whether the acquisition request was processed completely or not. See **ceStartReqStatus** on page 218.

Throws

ccPel::BadWindow

You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.

complete

```
PelBuffer complete (ccAcqFailure* result = 0,
  c_UInt32* appTag = 0, bool makeLocal = true,
  double maxWait = HUGE_VAL, bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Returns the image of the oldest outstanding acquisition and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one. Returns null if the acquisition failed.

Parameters

result — If not null, **result* is set to a description of why the acquisition failed. See **ccAcqFailure** on page 209.

appTag — If not null, **appTag* is set to the value that was passed to **start()**. If the acquisition failed because of too many outstanding acquisitions, or because acquisition was incomplete below then the returned **appTag* value is undefined. For trigger models that do not use **start()** to initiate the acquisition, this value is undefined.

makeLocal — If true, **baseComplete()** forces the image to be local (see **ccPelRoot::move()**). Otherwise, the image remains where it was acquired. For example, if your frame grabber (**ccFrameGrabber**) is also an accelerator (**ccAccelerator**), you might want to set *makeLocal* to false to leave the image on the accelerator to be processed there.

maxWait — The maximum number of seconds to wait for a complete acquisition to become available. The special value *HUGE_VAL* means to wait indefinitely. If the *maxWait* period elapses, **result->isIncomplete()** returns true.

autoStart — If *autoStart* is true and the selected trigger model associated with this FIFO allows **start()** to be invoked, then **start()** is invoked automatically when an acquisition completes. In this case, **start()** is invoked with an *appTag* of zero. If the acquisition is incomplete (**result->isIncomplete()** returns true) or if this function throws, **start()** is not invoked automatically.

startReqStatus If not NULL, returns whether the acquisition request was processed completely or not. See **ceStartReqStatus** on page 218.

Throws

ccPel::BadWindow

You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2763.

Typedefs

PelBuffer

```
typedef ccPelBuffer<ccPackedRGB32Pel> PelBuffer;
```

ccRLEBuffer

```
#include <ch_cvl/rlebuf.h>

class ccRLEBuffer : public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that contains a run-length encoded (RLE) image. Unlike an ordinary image, which is composed of a two-dimensional array of pixels, an RLE image is composed of a collection of *runs*. Each run has a pixel value and a length. RLE images are a compact method for storing images with regions of identical pixel values.

Unlike **ccPelBuffer**, the **ccRLEBuffer** class is not templated. Only images of **c_UInt8** pixels are supported.

To increase the efficiency of this class, a helper class, **cc_TmpRLEBuffer**, is provided. You should not create or manipulate **cc_TmpRLEBuffer** objects yourself.

Constructors/Destructors

ccRLEBuffer

```
ccRLEBuffer();

ccRLEBuffer(c_Int32 width, c_Int32 height,
  c_UInt8 background);

ccRLEBuffer(const ccRLEBuffer& rhs);

ccRLEBuffer(const cc_TmpRLEBuffer& rhs);
```

- `ccRLEBuffer();`

  Constructs a **ccRLEBuffer** with no pixels. This is also called a *degenerate* **ccRLEBuffer**.

- ```
  ccRLEBuffer(c_Int32 width, c_Int32 height,
    c_UInt8 background);
  ```

  Constructs a **ccRLEBuffer** with the specified dimensions and with every pixel set to the specified background value.
  ```

Parameters

width	The width of the image in pixels.
height	The height of the image in pixels.
background	The value to assign to all pixels in the image.

Notes

Both *width* and *height* must be greater than zero.

- `ccRLEBuffer(const ccRLEBuffer& rhs);`

  Copy constructor.

- `ccRLEBuffer(const cc_TmpRLEBuffer& rhs);`

  Copy constructor that takes a **cc_TmpRLEBuffer** as input.

Operators

operator= `ccRLEBuffer& operator= (const ccRLEBuffer&);`

Assignment operator.

operator= `ccRLEBuffer& operator= (const cc_TmpRLEBuffer& rhs);`

Assignment operator for assigning a **cc_TmpRLEBuffer** to a **ccRLEBuffer**.

operator== `bool operator== (const ccRLEBuffer&) const;`

Equality operator. Two **ccRLEBuffer**s are equal if their dimensions and the values of all their pixels are the same.

operator!= `bool operator!= (const ccRLEBuffer& rhs) const;`

See the description of the **operator==** member for the definition of inequality.

Enumerations

Shape

```
enum Shape
```

This enumeration defines the structuring element shapes for the grey-scale morphology operations supported by **ccRLEBuffer**.

Value	Meaning
eHoriz	Use a 1x3 horizontal structuring element.
eVert	Use a 1x3 vertical structuring element.
eSquare	Use a 1x3 square structuring element

Structures

ccRLEBufferRun

```
struct ccRLEBufferRun
  {
  c_UInt8 value;
  c_UInt8 length;
  friend ccArchive& operator|| (ccArchive&,
    const ccRLEBufferRun&);
  bool operator< (const ccRLEBufferRun&) const;
  bool operator== (const ccRLEBufferRun&) const;
  }
```

This structure defines a single run of pixels within a **ccRLEBuffer**.

Members

value The pixel value of this run.

length The length of this run, in pixels.

Public Member Functions

transfer

```
void transfer(ccRLEBuffer& rhs);
```

Assigns the contents of the supplied **ccRLEBuffer** to this **ccRLEBuffer**. This member is identical to **operator=**.

Parameters

rhs The the **ccRLEBuffer** to assign.

isDegenerate `bool isDegenerate() const;`

Returns *true* if this **ccRLEBuffer** has no pixels.

setUnbound `void setUnbound();`

Frees all the pixel storage associated with this **ccRLEBuffer**. Calling **isDegenerate()** after calling **setUnbound()** returns true.

hintNumRuns `void hintNumRuns(int hint);`

You can improve the efficiency with which an image is encoded by the **encode()** and **encodePercent()** functions by calling this function and providing the number of runs required to encode the image. The **encode()** and **encodePercent()** functions will pre-allocate the number of runs you specify and will guarantee that all the runs will be contiguous in memory.

The value you supply to this function is used for all subsequent calls to **encode()** and **encodePercent()**.

Parameters

hint — The number of runs to pre-allocate.

If you specify 0 for *hint*, then subsequent calls to **encode()** and **encodePercent()** are not guaranteed to produce runs contiguous in memory.

encode

```
void encode(const ccPelBuffer_const<c_UInt8>& image);

void encode(const ccPelBuffer_const<c_UInt8>& image,
 const cmStd vector<c_UInt8>& pmap);

void encode(const ccPelBuffer_const<c_UInt8>& image,
 c_UInt8 thresh, bool invert=false);

void encode(const ccPelBuffer_const<c_UInt8>& image,
 c_UInt8 loThresh, c_UInt8 hiThresh, int softness,
 bool invert=false);

void encode(const ccPelBuffer_const<c_UInt8>& image,
 const cmStd vector<c_UInt8>& preMap,
 const ccPelBuffer_const<c_UInt8>& thresh,
 const cmStd vector<c_UInt8>& postMap);
```

- `void encode(const ccPelBuffer_const<c_UInt8>& image);`

  Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

  Parameters

  image The image to run-length encode.

  Notes

  Any client coordinate system associated with *image* is preserved in this **ccRLEBuffer**.

- `void encode(const ccPelBuffer_const<c_UInt8>& image, const cmStd vector<c_UInt8>& pmap);`

  Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

  As the image is encoded, the supplied pixel map is applied to each pixel in the input image.

  Parameters

  image The image to run-length encode.

  pmap The pixel map to apply to the input image. Each pixel in the input image is used as an index into *pmap*; the input pixel value is replaced with the value from *pmap*.

  pmap must be at least as large as the largest pixel value in *image*.

Notes

Any client coordinate system associated with *image* is preserved in this **ccRLEBuffer**.

- ```
  void encode(const ccPelBuffer_const<c_UInt8>& image,
    c_UInt8 thresh, bool invert=false);
  ```

  Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

  As the image is encoded, the supplied hard threshold is used to segment the image.

  Parameters

image	The image to run-length encode
thresh	The threshold value. All pixels in *image* with values greater than or equal to *thresh* are set to 1; pixels with values less than *thresh* are set to 0.
invert	If *invert* is *true*, then all pixels in *image* with values greater than or equal to *thresh* are set to 0 and pixels with values less than *thresh* are set to 1. If *invert* is *false*, then all pixels in *image* with values greater than or equal to *thresh* are set to 1 and pixels with values less than *thresh* are set to 0.

- ```
  void encode(const ccPelBuffer_const<c_UInt8>& image,
    c_UInt8 loThresh, c_UInt8 hiThresh, int softness,
    bool invert=false);
  ```

  Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

  As the image is encoded, the supplied thresholds are used to segment the image.

  Parameters

image	The image to run-length encode
loThresh	The low threshold value. All pixels in *image* with values less than *loThresh* are set to 0.
hiThresh	The high threshold value. All pixels in *image* with values greater than or equal to *hiThresh* are set to *softness+1*.
softness	The number of intermediate pixel weight ranges into which the image will be segmented. *softness* evenly spaced ranges of pixel values are defined. Each range of pixel values is assigned a pixel weight between 0 and *softness*, with the pixel weights distributed

in a linear fashion. Each pixel in the input image with a value between *loThresh* and *hiThresh* is assigned the pixel weight associated with the range into which the pixel's value falls.

invert — *invert* defines the polarity of the image. If *invert* is *false*, *loThresh* and *hiThresh* are interpreted as described above. If *invert* is *true*, pixels with values less than *loThresh* are assigned a weight of *softness*+1 while pixels with values greater than *hiThresh* are assigned a weight of 0. For images of dark objects on a light background, *invert* should be *false*.

Notes

Any client coordinate system associated with *image* is preserved in this **ccRLEBuffer**.

- ```
  void encode(const ccPelBuffer_const<c_UInt8>& image,
    const cmStd vector<c_UInt8>& preMap,
    const ccPelBuffer_const<c_UInt8>& thresh,
    const cmStd vector<c_UInt8>& postMap);
  ```

Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

As the image is encoded, the supplied threshold image, threshold value, and pixel maps are used to segment the image.

Parameters

image — The image to encode.

preMap — The pixel map to apply to the input image. Each pixel in the input image is used as an index into *preMap*; the input pixel value is replaced with the value from *preMap*.

preMap must be at least as large as the largest pixel value in *image*.

thresh — The threshold image. *thresh* must have the same dimensions as *image*. The value of each pixel in *thresh* is subtracted from the corresponding pixel in *image*, after *image* is mapped by *preMap*. The value that results from this operation is mapped using *postMap*.

postMap — The pixel map to apply to the result of the image subtraction. Each pixel value that results from the subtraction of the *thresh* from *image* is used as an index into *postMap*; the pixel value in the encoded image set to the value from *postMap*.
  ```

postMap must be at least as large as the largest difference in pixel values between *image* and *thresh*.

Notes

Any client coordinate system associated with *image* is preserved in this **ccRLEBuffer**.

encodePercent

```
void encodePercent(
 const ccPelBuffer_const<c_UInt8> &image,
 int lowTailPercent, int highTailPercent,
 int threshPercent, bool invert = false);

void encodePercent(
 const ccPelBuffer_const<c_UInt8> &image,
 int lowTailPercent, int highTailPercent,
 int threshLowPercent, int threshHighPercent,
 int softness, bool invert = false);
```

-
```
void encodePercent(
 const ccPelBuffer_const<c_UInt8> &image,
 int lowTailPercent, int highTailPercent,
 int threshPercent, bool invert = false);
```

Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

As the image is encoded, the supplied histogram tail percentages and relative threshold value are used to segment the image. The segmentation is performed by discarding the specified percentages of low and high tail pixels, then setting the threshold value at the specified percentage of the remaining pixels.

Parameters

image — The image to encode.

lowTailPercent — The lower *lowTailPercent* of all pixels in *image* are discarded before computing the threshold value. This value must be greater than or equal to 0 and less than *highTailPercent*.

highTailPercent — The upper 100-*highTailPercent* of all pixels in *image* are discarded before computing the threshold value. This value must be less than or equal to 100 and greater than *lowTailPercent*.

threshPercent — After discarding the pixels specified by *lowTailPercent* and *highTailPercent*, the threshold value is computed to be the value of the pixel below which *threshPercent* of all remaining pixels in *image* lie.

invert If *invert* is *false* (the default), all pixels in *image* with values greater than or equal to the computed threshold value are set to 1 and all pixels in *image* with values less than the computed threshold value are set to 0. If *invert* is *true*, all pixels in *image* with values greater than or equal to the computed threshold value are set to 0 and all pixels in *image* with values less than the computed threshold value are set to 1.

- ```
  void encodePercent(
    const ccPelBuffer_const<c_UInt8> &image,
    int lowTailPercent, int highTailPercent,
    int threshLowPercent, int threshHighPercent,
    int softness, bool invert = false);
  ```

  Sets the contents of this **ccRLEBuffer** to be the run-length encoded equivalent of the supplied **ccPelBuffer_const<c_UInt8>**.

  As the image is encoded, the supplied histogram tail percentages and relative threshold value are used to segment the image. The segmentation is performed by discarding the specified percentages of low and high tail pixels, then setting the low and high threshold values at the specified percentages of the remaining pixels.

  Parameters

  image The image to encode.

  lowTailPercent The lower *lowTailPercent* of all pixels in *image* are discarded before computing the threshold values. This value must be greater than or equal to 0 and less than *highTailPercent*.

  highTailPercent The upper 100-*highTailPercent* of all pixels in *image* are discarded before computing the threshold values. This value must be less than or equal to 100 and greater than *lowTailPercent*.

  threshLowPercent

  After discarding the pixels specified by *lowTailPercent* and *highTailPercent*, the low threshold value is computed to be the value of the pixel below which *threshLowPercent* of all remaining pixels in *image* lie. This value must be greater than or equal to 0 and less than *threshHighPercent*.

  threshHighPercent
  ```

After discarding the pixels specified by *lowTailPercent* and *highTailPercent*, the high threshold value is computed to be the value of the pixel above which *threshHighPercent* of all remaining pixels in *image* lie. This value must be less than or equal to 100 and greater than *threshLowPercent*

softness — The number of intermediate pixel weight ranges into which the image will be segmented. *softness* evenly spaced ranges of pixel values between the computed low and high thresholds are defined. Each range of pixel values is assigned a pixel weight between 0 and *softness*, with the pixel weights distributed in a linear fashion. Each pixel in the input image with a value between the computed low threshold and the computed high threshold is assigned the pixel weight associated with the range into which the pixel's value falls.

invert — *invert* defines the polarity of the image. If *invert* is *false*, the computed low and high thresholds are interpreted as described above. If *invert* is *true*, pixels with values less than the computed low threshold are assigned a weight of *softness*+1 while pixels with values greater than or equal to the computed high threshold are assigned a weight of 0. For images of dark objects on a light background, *invert* should be *false*.

isBinary

```
bool isBinary() const;
```

Returns *true* if this **ccRLEBuffer** was encoded using a hard binary threshold.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

createTime

```
double createTime() const;
```

Returns the amount of time in milliseconds that were required to create or modify this **ccRLEBuffer**.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

Notes

This function might return a value of 0 if the current platform cannot provide accurate and inexpensive timing information.

isContiguous

```
bool isContiguous() const;
```

Returns *true* if all of the runs that make up this **ccRLEBuffer** are stored in a single contiguous block of memory.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

makeContiguous

```
void makeContiguous();
```

This function reallocates memory as needed, then stores all of the runs that make up this **ccRLEBuffer** in a single contiguous block of memory. If all of the runs that make up this **ccRLEBuffer** are already stored in a single contiguous block of memory, this function has no effect.

Notes

createTime() will return the amount of time that was required by this function.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

offset

```
ccIPair offset() const;
void offset(c_Int32 left, c_Int32 top);
void offset(const ccIPair&);
```

- `ccIPair offset() const;`

  Returns a **ccIPair** which gives the image coordinates of this **ccRLEBuffer**'s upper left corner.

  Throws

  ccRLEBuffer::Degenerate
  This **ccRLEBuffer** has no pixels.

- `void offset(c_Int32 left, c_Int32 top);`

  Sets this **ccRLEBuffer**'s image coordinate offset. Calling this function has no effect on the contents of this **ccRLEBuffer**, nor on any client coordinate system associated with it.

  Parameters

  left The x-coordinate of the offset

top The y-coordinate of the offset

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

- `void offset(const ccIPair& offset);`

Sets this **ccRLEBuffer**'s image coordinate offset. Calling this function has no effect on the contents of this **ccRLEBuffer**, nor on any client coordinate system associated with it.

Parameters

offset The offset

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

clientFromImageXform

```
cc2Xform clientFromImageXform() const;

void clientFromImageXform(const cc2Xform& xform);
```

- `cc2Xform clientFromImageXform() const;`

Returns the **cc2Xform** that transforms points in this **ccRLEBuffer**'s image coordinate system to the client coordinate system.

- `void clientFromImageXform(const cc2Xform& xform);`

Sets the **cc2Xform** that transforms points in this **ccRLEBuffer**'s image coordinate system to the client coordinate system.

Parameters

xform The **cc2Xform**

Notes

Setting either the image to client **cc2Xform** or the client to image **cc2Xform** also sets the other **cc2Xform**.

imageFromClientXform

```
cc2Xform imageFromClientXform() const;
void imageFromClientXform(const cc2Xform& xform);
```

```
cc2Xform imageFromClientXform() const;
```

Returns the **cc2Xform** that transforms points in this **ccRLEBuffer**'s client coordinate system to the image coordinate system.

- `void imageFromClientXform(const cc2Xform& xform);`

  Sets the **cc2Xform** that transforms points in this **ccRLEBuffer**'s client coordinate system to the image coordinate system.

  Parameters

  xform The **cc2Xform**

  Notes

  Setting either the image to client **cc2Xform** or the client to image **cc2Xform** also sets the other **cc2Xform**.

copyXforms

```
void copyXforms(const ccRLEBuffer& rlebuf);
void copyXforms(const cc_PelBuffer& pelbuf);
```

- `void copyXforms(const ccRLEBuffer& rlebuf);`

  Sets the **cc2Xform**s associated with this **ccRLEBuffer** to be those associated with the supplied **ccRLEBuffer**. Both the **cc2Xform** that transforms points in this **ccRLEBuffer**'s image coordinate system to the client coordinate system and the **cc2Xform** that transforms points in the client coordinate system to this **ccRLEBuffer**'s image coordinate system are copied from the supplied **ccRLEBuffer**.

  Cognex recommends that you use this function rather than **imageFromClientXform()** and **clientFromImageXform()**.

  Parameters

  rlebuf The **ccRLEBuffer** from which to copy the two **cc2XForm**s.

- `void copyXforms(const cc_PelBuffer& pelbuf);`

  Sets the **cc2Xform**s associated with this **ccRLEBuffer** to be those associated with the supplied **cc_PelBuffer**. Both the **cc2Xform** that transforms points in this **ccRLEBuffer**'s image coordinate system to the client coordinate system and the **cc2Xform** that transforms points in the client coordinate system to this **ccRLEBuffer**'s image coordinate system are copied from the supplied **ccPelBuffer**.

  Cognex recommends that you use this function rather than **imageFromClientXform()** and **clientFromImageXform()**.

  Parameters

  pelbuf The **cc_PelBuffer** from which to copy the two **cc2XForm**s

---

image

```
ccPelBuffer<c_UInt8> image() const;

void image(ccPelBuffer<c_UInt8>& answer) const;
```

- `ccPelBuffer<c_UInt8> image() const;`

  Returns a **ccPelBuffer<c_UInt8>** that is identical in size and in the value of every pixel to this **ccRLEBuffer**.

  Notes

  If this **ccRLEBuffer** contains no pixels, then the returned **ccPelBuffer<c_UInt8>** is constructed by the default **ccPelBuffer<c_UInt8>** constructor.

- `void image(ccPelBuffer<c_UInt8>& answer) const;`

  Copies a **ccPelBuffer<c_UInt8>** that is identical in size and in the value of every pixel to this **ccRLEBuffer** to the supplied **ccPelBuffer<c_UInt8>** reference.

  Parameters

  answer A reference to a **ccPelBuffer<c_UInt8>** into which the image is placed.

  Notes

  If this **ccRLEBuffer** contains no pixels, then the returned **ccPelBuffer<c_UInt8>** is as constructed by the default **ccPelBuffer<c_UInt8>** constructor.

width

```
c_Int32 width() const;
```

Returns the width in pixels of this **ccRLEBuffer**.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

height `c_Int32 height() const;`

Returns the height in pixels of this **ccRLEBuffer**.

Throws
ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

size `ccIPair size() const;`

Returns the height and width in pixels of this **ccRLEBuffer**. The height is returned as **size().y()**; the width is returned as **size().x()**.

Throws
ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

pointToRow `const ccRLEBufferRun* pointToRow(c_Int32 rowNum) const;`

Returns a pointer to the first **ccRLEBufferRun** of the specified row of this **ccRLEBuffer**. All of the **ccRLEBufferRun**s that make up a single row of a **ccRLEBuffer** are guaranteed to be contiguous in memory.

The end of every row in a **ccRLEBuffer** is marked by a **ccRLEBufferRun** with a length of zero.

ccRLEBufferRuns in different rows of a **ccRLEBuffer** are guaranteed to be contiguous only if **isContiguous()** returns *true*.

Parameters
rowNum The row number

Notes
The pointer returned by this function is only valid until the next non-**const** member of this **ccRLEBuffer** is called.

Row numbers are specified in image coordinates; the first row of runs should be accessed as shown below:

```
this->pointToRow(this->offset().y())
```

Throws
ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

ccPel::BadCoord
Invalid row number.

pointToRat

```
const ccRLEBufferRun* const * pointToRat() const;
```

Returns a pointer to the row address table (RAT). The returned pointer points to the beginning of a contiguous array of pointers to **ccRLEBufferRun**. Each element in the array points to the first run of the corresponding row of this **ccRLEBuffer**.

All rows have at least two **ccRLEBufferRun** objects (each row is terminated with a zero-length **ccRLEBufferRun**).

Notes

The returned pointers are only valid until the next call to a non-**const** function in this **ccRLEBuffer**.

getPel

```
c_UInt8 getPel(const ccIPair& location) const;
c_UInt8 getPel(c_Int32 x, c_Int32 y) const;
```

- `c_UInt8 getPel(const ccIPair& location) const;`

Returns the value of the pixel at the specified location within this **ccRLEBuffer**.

Parameters

location A **ccIPair** containing the location of the pixel to get

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

- `c_UInt8 getPel(c_Int32 x, c_Int32 y) const;`

Returns the value of the pixel at the specified location within this **ccRLEBuffer**.

Parameters

x The x-coordinate of the pixel to get

y The y-coordinate of the pixel to get

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

subWindow

```
cc_TmpRLEBuffer subWindow(const ccPelRect& theRect) const;
```

Returns a **cc_TmpRLEBuffer** representing a subwindow, specified by the supplied **ccPelRect**, of this **ccRLEBuffer**.

The offset of the subwindow will be equal to the offset of this **ccRLEBuffer** plus the values specified for the origin of the supplied **ccPelRect**.

The client coordinate system transformation of the returned **cc_TmpRLEBuffer** is the same as that of this **ccRLEBuffer**.

Parameters

theRect A rectangle defining the region of this **ccRLEBuffer** to return.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

ccPel::BadWindow
theRect is not contained entirely within this **ccRLEBuffer** or *theRect* has either a width or height of zero.

Notes

The **cc_TmpRLEBuffer** type can be efficiently copied or assigned to a **ccRLEBuffer**.

mask

```
cc_TmpRLEBuffer mask(const ccRLEBuffer& mask,
  c_UInt8 match, c_UInt8 replace, bool invert=false) const;
```

Returns a **cc_TmpRLEBuffer** that is a masked version of this **ccRLEBuffer**. The masked image is constructed as follows: Each pixel in this **ccRLEBuffer** that corresponds to a pixel in *mask* which has the same value as *match* is replaced with a pixel of with the value *replace*. If *invert* is *true*, then each pixel in this **ccRLEBuffer** that does not correspond to a pixel in *mask* which has the same value as *match* is replaced with a pixel of the value *replace*.

Parameters

mask A **ccRLEBuffer** containing the mask image.

match The match value. Pixels in this **ccRLEBuffer** which correspond to pixels in *mask* with a value of *match* are replaced with a value of *replace.*

replace The replacement value.

invert If *invert* is true, then pixels this **ccRLEBuffer** which do not correspond to pixels in *mask* with a value of *match* are replaced with a value of *replace*.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels, or *mask* has no pixels.

Notes

The **cc_TmpRLEBuffer** type can be efficiently copied or assigned to a **ccRLEBuffer**.

The mask image must have the same offset and dimensions as this **ccRLEBuffer**.

combine

```
cc_TmpRLEBuffer combine(ccIPair offset,
  const ccRLEBuffer& subImage,
  const cmStd vector<c_UInt8>& pmap, c_UInt8 passVal=0)
  const;
```

Returns a **cc_TmpRLEBuffer** produced by combining the supplied **ccRLEBuffer** with this one. The **ccRLEBuffer**s are combined by overwriting part or all of this **ccRLEBuffer** with the contents of the supplied **ccRLEBuffer**. *offset* specifies where in this **ccRLEBuffer**'s image coordinate system the origin of *subImage* is placed. If *subImage* has an offset, it is respected as well.

You can specify a pixel map to apply to *subImage*, and you can specify that pixels in this **ccRLEBuffer** that correspond to pixels in the mapped *subImage* with a specified value not be replaced.

Parameters

offset — The location in this **ccRLEBuffer** at which to place the contents of the supplied **ccRLEBuffer**.

subImage — The **ccRLEBuffer** to copy into this **ccRLEBuffer**.

pmap — The pixel map to apply to *subImage*. Each pixel in *subImage* is replaced with the value at the corresponding offset in *pmap*.

passVal — Only pixels in this **ccRLEBuffer** which correspond to pixels in *subImage* that do not have a value of *passVal* (after the mapping operation) are replaced with pixels from *subImage*.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels, or *subImage* has no pixels.

Notes

The **cc_TmpRLEBuffer** type can be efficiently copied or assigned to a **ccRLEBuffer**.

map

```
cc_TmpRLEBuffer map(const cmStd vector<c_UInt8>& map)
  const;
```

Returns a mapped version of this **ccRLEBuffer**. Every pixel in this **ccRLEBuffer** is replaced with the pixel value found at the pixel's offset within the supplied pixel map

Parameters

map A vector of **c_UInt8** that contains the pixel map. *map* must contain at least as many elements as the largest pixel value in this **ccRLEBuffer**.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

Notes

The **cc_TmpRLEBuffer** type can be efficiently copied or assigned to a **ccRLEBuffer**.

---

histo

```
void histo(cmStd vector<c_UInt32>& histo) const;

void histo(cmStd vector<c_UInt32>& histoPels,
  cmStd vector<c_UInt32>& histoRuns) const;
```

- ```
  void histo(cmStd vector<c_UInt32>& histo) const;
  ```

  Calculates a histogram of the distribution of pixel values in this **ccRLEBuffer**.

  Parameters

  histo A reference to a vector of **c_UInt32** in which the histogram is placed. This vector must contain at least as many elements as the largest pixel value in this **ccRLEBuffer**.

  Notes

  The supplied vector of **c_UInt32** is *not* initialized by this function.

- ```
  void histo(cmStd vector<c_UInt32>& histoPels,
    cmStd vector<c_UInt32>& histoRuns) const;
  ```

  Computes two histograms of the distribution of pixel values in this **ccRLEBuffer**. The first histogram is a simple histogram of the pixel values in this **ccRLEBuffer**. The second histogram is of the distribution of runs of pixels of each pixel value.

  For example, if this **ccRLEBuffer** contained five runs with a pixel value of 27, and each of these runs had a length of 50, the value of the 27th element of the first histogram would be 250 while the value of the 27th element of the second histogram would be 5.

Parameters

histoPels A reference to a vector of **c_UInt32** in which the histogram of pixel values is placed. This vector must contain at least as many elements as the largest pixel value in this **ccRLEBuffer**.

histoRuns A reference to a vector of **c_UInt32** in which the histogram of runs is placed. This vector must contain at least as many elements as the largest pixel value in this **ccRLEBuffer**.

Notes

The supplied vectors of **c_UInt32** are *not* initialized by this function.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

gmorphMin

```
cc_TmpRLEBuffer gmorphMin(Shape=eSquare) const;
```

Performs a grey-scale morphological minimize operation using the specified structuring element.

Parameters

Shape The shape of the structuring element to apply. *shape* must be one of the following values:

ccRLEBuffer::eHoriz
ccRLEBuffer::eVert
ccRLEBuffer::eSquare

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

Notes

At image borders, where full neighborhoods cannot be formed, partial neighborhoods are used. The returned **cc_TmpRLEBuffer** has the same dimensions as the source.

The **cc_TmpRLEBuffer** type can be efficiently copied or assigned to a **ccRLEBuffer**.

gmorphMax

```
cc_TmpRLEBuffer gmorphMax(Shape=eSquare) const;
```

Performs a grey-scale morphological maximize operation using the specified structuring element.

Parameters

Shape The shape of the structuring element to apply. *shape* must be one of the following values:

ccRLEBuffer::eHoriz
ccRLEBuffer::eVert
ccRLEBuffer::eSquare

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

Notes

At image borders, where full neighborhoods cannot be formed, partial neighborhoods are used. The returned **cc_TmpRLEBuffer** has the same dimensions as the source.

The **cc_TmpRLEBuffer** type can be efficiently copied or assigned to a **ccRLEBuffer**.

numRuns

```
int numRuns() const;
int numRuns(c_UInt8 val) const;
```

- `int numRuns() const;`

Return the number of runs in this **ccRLEBuffer**.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

- `int numRuns(c_UInt8 val) const;`

Return the number of runs in this **ccRLEBuffer** that have the supplied pixel value.

Parameters

val The pixel value.

Throws

ccRLEBuffer::Degenerate
This **ccRLEBuffer** has no pixels.

avgRunLength

```
double avgRunLength() const;
double avgRunLength(c_UInt8 val) const;
```

- `double avgRunLength() const;`

  Return the average run length of the runs in this **ccRLEBuffer**.

  Throws

  ccRLEBuffer::Degenerate
  This **ccRLEBuffer** has no pixels.

- `double avgRunLength(c_UInt8 val) const;`

  Return the average run length of the runs in this **ccRLEBuffer** that have the supplied pixel value.

  Parameters

  val The pixel value.

  Throws

  ccRLEBuffer::Degenerate
  This **ccRLEBuffer** has no pixels.

ccRoiProp

```
#include <ch_cvl/prop.h>

class ccRoiProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The region of interest (ROI) property lets you specify which part of the acquired image you wish to process. The acquisition FIFO uses the ROI property to optimize acquisition time. For line scan cameras, you use the ROI property to refine the height and width of the acquired image.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Depending on the capabilities of the video hardware, the ROI setting may affect which pixels are actually written in the pixel root image, allowing the acquisition FIFO to acquire a smaller number of pixels, thereby reducing acquisition time.

For example, the following code fragments are equivalent, but the second fragment may be optimized:

```
   // Never optimized
fifo->start();
pb = fifo->complete();
pb.subWindow(x,y,w,h);

   // May be optimized
fifo->properties().roi(ccPelRect(x,y,w,h));
fifo->start();
pb = fifo->complete();
```

Pixels outside the ROI have unspecified values. The ROI setting does not affect the root image size. The resulting pel buffer window size is the intersection of the ROI and the whole image; in other words, an ROI larger than the image is clipped to the root image size. A null rectangle (see **ccPelRect::isNull()**) specifies the entire image.

For line scan cameras, the ROI property specifies the height and width of the acquired image. The height and width must satisfy the following three conditions. Assume that **videoFormat.width()** is the width of the video format (see **width()** on page 3400), *width* and *height* are the desired width and height of the image, and *xOffset* is the x-offset.

- 1 <= width <= **videoFormat.width()** - xOffset
- 1 <= height <= 8192
- width x height <= 4194304 (4 x 1024 x 1024)

Enumerations

Mode

```
enum Mode;
```

This enumeration defines the automatic ROI behavior.

Value	Meaning
eAutomatic = 0	The acquisition software will adjust the ROI to the capabilities of the hardware. If the actual ROI is larger than the requested ROI, a window of the requested size is applied to each acquired image.
eManual = 1	The ROI setting is written to the hardware as given and no adjustment is made to the acquired image. *eManual* should only be used in specific circumstances where *eAutomatic* does not work correctly. For example, a line scan camera that acquires lines while a trigger signal is active, such that the height of the image is determined by the trigger signal and not the ROI setting.

Constructors/Destructors

ccRoiProp

```
ccRoiProp();
explicit ccRoiProp(const ccPelRect& roi);
```

- `ccRoiProp();`

  Creates a new ROI property not associated with any FIFO. The region of interest is the entire image.

- `explicit ccRoiProp(const ccPelRect& roi);`

  Creates a new ROI property not associated with any FIFO. The region of interest is specified by the rectangle *roi*.

  Parameters

  roi The region of interest. A null ROI means the entire image.

Public Member Functions

roi

```
const ccPelRect& roi() const;
void roi(const ccPelRect& r);
```

- `const ccPelRect& roi();`

  Returns the current region of interest.

  Notes

  If the region of interest was never set in the constructor or with the setter, this function returns a null ROI. Once you set an ROI, this function and **actualRoi()** return the same value.

- `void roi(const ccPelRect& roi);`

  Sets the region of interest to the rectangle specified by *roi*.

  Parameters

  roi The region of interest. A null ROI means the entire image.

Notes

If you specify a region of interest, the **window()** of the pel buffer returned by **complete()** or **baseComplete()** is the intersection of the ROI and the entire image. If this intersection is null, **complete()** or **baseComplete()** will throw *ccPel::BadWindow*.

Throws

ccRoiProp::BadParams
roi.ul().x(), **roi.ul().y()**, **roi.height()**, or **roi.width()** were negative.

actualRoi

```
ccPelRect actualRoi() const;
```

Returns the actual region of interest for the FIFO associated with this property even if the setter **roi()** has never been called. If the region of interest is the entire image, the returned rectangle is the size of the image, not the null rectangle.

Once the setter has been called, **roi()** and **actualRoi()** return the same value.

Throws

ccRoiProp::NoFrameGrabber
This property is not associated with an acquisition FIFO.

roiMode

```
ccRoiProp::Mode roiMode() const;

void roiMode(ccRoiProp::Mode mode);
```

- `ccRoiProp::Mode roiMode() const;`

Gets the ROI mode for this FIFO. The returned value is one of

ccRoiProp::eAutomatic
ccRoiProp::eManual

Throws

ccRoiProp::NoFrameGrabber
This property is not associated with an acquisition FIFO.

- `void roiMode(ccRoiProp::Mode mode);`

Sets the ROI mode for this FIFO.

Parameters

mode The mode to set. *mode* must be one of

ccRoiProp::eAutomatic
ccRoiProp::eManual

Throws

ccRoiProp::BadParams

mode is not supported on the hardware associated with this FIFO.

isModeSupported

```
bool isModeSupported(ccRoiProp::Mode mode);
```

Returns true if the specified mode can be set on this FIFO, false if it cannot.

Parameters

mode The mode to test. *mode* must be one of

ccRoiProp::eAutomatic
ccRoiProp::eManual

ccRSIDefs

```
#include <ch_cvl/rsi.h>

class ccRSIDefs
```

A name space that holds enumerations and constants used with the RSI Search tool. See **ccRSIModel**.

Enumerations

DOF

`enum DOF`

This enumeration defines constants that specify degrees of freedom at train time or at search time.

Value	Meaning
eAngle = 0x1	Consider rotation.
eUniformScale = 0x2	Consider uniform scaling along both the x- and y-axis.
eXScale = 0x4	Consider scaling along the x-axis.
eYScale = 0x8	Consider scaling along the y-axis.
eShear = 0x10	Consider shear.

Method

`enum Method`

This enumeration defines constants that specify the method used to compute the sub-pixel model pose.

Value	Meaning
eStandard = 1	A basic mathematical model is used to estimate the sub-pixel position with the highest score. This model may not accuractly model changes in the correlation score at small image offsets.
eHighAccuracy = 2	A mathematical model that better characterizes how the normalized correlation coefficient changes at small sub-pixel image offsets is used.
kDefaultMode	The default mode (*eStandard)*.

Mode

`enum Mode`

This enumeration defines constants that specify when the template images are generated.

Value	Meaning
eGenerateTemplates = 0x1	Templates are generated at training time, based on the specified DOF range or nominal values.
eUnrestrictedDOFs = 0x2	Templates are generated at run time.
kDefaultMode	The default mode (*eGenerateTemplates)*.

GranularityGenerator

`enum GranularityGenerator`

This enumeration defines constants that specify how to generate the coarse granularity model.

Value	Meaning
eGranularityGeneratorRobust = 0x1	Recommended for most applications. Generates a coarse model that will find all model instances.
eGranularityGeneratorFast = 0x2	Generates a coarse model that provides faster execution at run time. For some model images, particularly those with all fine features, using this generator may cause the tool to fail to find some model instances.
kDefaultGranularityGenerator	The default mode (*eGranularityGeneratorRobust)*.

Compression

`enum Compression`

This enumeration defines constants that specify what type of compression to apply to the trained model.

Value	Meaning
eCompressionNone = 0x1	Do not compress the trained model.
eCompressionLossy = 0x2	Compress the trained model using a lossy compression method. This method produces the fastest search, but may miss some model instances.
eCompressionLossless = 0x4	Compress the trained model using a lossless compression method. This method may result in a faster search than using *eCompressionNone*, but without the risk of failing to find some instances.
kDefaultCompression	The default mode (*eCompressionNone)*.

CombineDOF

`enum CombineDOF`

This enumeration defines constants that specify whether a non-identity generalized model origin (**ccRSIModel::trainClientFromModel()**) is considered to have been combined with training-time DOF nominal or range values. This enumeration only applies when training-time template generation is specified *(***ccRSITrainParams::mode()** *is eGenerateTemplates).*

Value	Meaning
eCombineDOFTrainClientFromModel = 0x1	The DOFs are considered to have been combined with the generalized origin.
eCombineDOFIdentity = 0x2	The DOFs are not considered to have been combined.
kDefaultCombineDOF	The default combination mode (*eCombineDOFTrainClientFromModel)*.

DrawMode `enum DrawMode`

This enumeration defines types of result graphics you can draw for RSI results

Value	Meaning
eDrawOrigin = 0x1	Draws a cross at the location of the result. The cross is aligned to the angle at which the pattern is found.
eDrawLabel = 0x2	Draws a text label at the result location.
eDrawBoundingBox = 0x4	Draws a box around the region of the image that corresponds to the matched model.
eDrawStandard = 0x7	Draws a labeled, pattern-aligned cross at the pattern location along with a box that corresponds to the matched model.

ccRSIModel

```
#include <ch_cvl/rsi.h>

class ccRSIModel
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains the trained representation of an RSI model as well as the functions to do the training and to search an image for the model.

Constructors/Destructors

ccRSIModel

```
ccRSIModel();

~ccRSIModel();
```

- `ccRSIModel();`

  Construct an untrained RSI alignment model with default values.

- `~ccRSIModel();`

  Destructor.

Public Member Functions

origin

```
cc2Vect origin() const;

void origin(const cc2Vect &origin);
```

- `cc2Vect origin() const;`

  Returns the origin of the model in the client coordinate system. You can get the origin whether the model is trained or untrained.

- `void origin(const cc2Vect &origin);`

  Sets the origin of the model in the client coordinate system. You can set the origin whether the model is trained or untrained.

  The origin is associated with the model, but may reside outside of the model. When an instance of the model is found in the search image, the result location is defined by the position of the origin in the search image.

  Parameters

  origin The origin of the model. Default **cc2Vect(**0,0**)**.

  Note Setting **trainClientFromModel()** changes the origin. Setting the origin changes the translation component of **trainClientFromModel()** but not the matrix portion of the transform.

  Setting the origin affects both the **location()** and the **pose()**.

trainClientFromModel

```
cc2Xform trainClientFromModel() const;

virtual void trainClientFromModel(const cc2Xform& xform);
```

- `cc2Xform trainClientFromModel() const;`

  Get the transform that maps from model coordinates to the training image client coordinates.

- `virtual void trainClientFromModel(const cc2Xform&);`

  Set the transform that maps from model coordinates to the training image client coordinates.

  Model coordinates are a client-controlled coordinate system that specifies how RSI search is to match the model to an image and report a result pose. A result pose maps from model coordinates to runtime client coordinates. The specified degrees of freedom define the search range as a mapping from model coordinates to runtime coordinates.

  Model coordinates include the model's origin as its translation component, and setting the model coordinates with this function always sets the origin.

  Training never modifies model coordinates. They are relative to the client coordinates established by the most recent training.

  Note You can set the **trainClientFromModel()** transform only before training, To change the **trainClientFromModel()** transform on a trained model, you must first call **untrain()** and then train again.

Parameters

xform The transform used to perform the mapping. Default is the identity transform.that model coordinates includes the

Throws

ccRSIDefs::AlreadyTrained
The model is already trained.

---

train

```
void train(const ccPelBuffer_const<c_UInt8>& image,
                  const ccRSITrainParams &trainParams,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0);

void train(const cc3PlanePelBuffer_const& image,
                  const ccRSITrainParams& trainParams,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0);

void train(const ccPelBuffer_const<c_UInt8>& image,
                  const ccPelBuffer_const<c_UInt8>& mask,
                  const ccRSITrainParams &trainParams,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0);

void train(const cc3PlanePelBuffer_const& image,
                  const ccPelBuffer_const<c_UInt8>& mask,
                  const ccRSITrainParams& trainParams,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0);
```

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
                    const ccRSITrainParams &trainParams,
                    ccDiagObject* diagobj=0,
                    c_UInt32 diagFlags=0);
  ```

Trains this model from the given 8-bit image using the specified training parameters. If the model is already trained, it is automatically untrained and retrained.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

Parameters

image The 8-bit training image.

trainParams The training parameters

diagobj A diagnostic object

diagFlags Diagnostic flags
  ```

Throws

ccRSIDefs::BadImageContent
The Image is featureless.

ccRSIDefs::BadImage
image is unbound or has a null window

ccRSIDefs::NotImplemented
The image client coordinate system is nonlinear

ccRSIDefs::BadImageSize
The image is too small.

ccTimeout::Expired
The operation timed out.

- ```
  void train(const cc3PlanePelBuffer_const& image,
                  const ccRSITrainParams& trainParams,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0);
  ```

Trains this model from the given three-plane (R,G,B) image using the specified training parameters. If the model is already trained, it is automatically untrained and retrained.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

Parameters

image The three-plane training image.

trainParams The training parameters

diagobj A diagnostic object

diagFlags Diagnostic flags

Throws

ccRSIDefs::BadImageContent
The Image is featureless.

ccRSIDefs::BadImage
image is unbound or has a null window

ccRSIDefs::NotImplemented
The image client coordinate system is nonlinear

ccRSIDefs::BadImageSize
The image is too small.
  ```

ccTimeout::Expired
The operation timed out.

- ```
  void train(const ccPelBuffer_const<c_UInt8>& image,
                   const ccPelBuffer_const<c_UInt8>& mask,
                   const ccRSITrainParams &trainParams,
                   ccDiagObject* diagobj=0,
                   c_UInt32 diagFlags=0);
  ```

Trains this model from the given 8-bit image and mask using the specified training parameters. If the model is already trained, it is automatically untrained and retrained.

This function respects CVL timeouts. If a timeout occurs, the model is not trained.

Parameters

image The 8-bit training image.

mask The image mask. Pixels whose value is 0 indicate “don’t care” pixels in the training image; pixels whose value is 255 indicate “care” pixels in the training image.

trainParams The training parameters

diagobj A diagnostic object

diagFlags Diagnostic flags

Throws

ccRSIDefs::BadImageContent
The Image is featureless.

ccRSIDefs::BadImage
image is unbound or has a null window
mask is unbound
mask does not have the same window as image
mask pels have values other than 0 or 255

ccRSIDefs::NotImplemented
The image client coordinate system is nonlinear

ccRSIDefs::BadImageSize
The image is too small.

ccTimeout::Expired
The operation timed out.
  ```

- ```
  void train(const cc3PlanePelBuffer_const& image,
                   const ccPelBuffer_const<c_UInt8>& mask,
                   const ccRSITrainParams& trainParams,
                   ccDiagObject* diagobj=0,
                   c_UInt32 diagFlags=0);
  ```

  Trains this model from the given three-plane (R,G,B) image and mask using the specified training parameters. If the model is already trained, it is automatically untrained and retrained.

  This function respects CVL timeouts. If a timeout occurs, the model is not trained.

  Parameters

  image The three-plane training image.

  mask The image mask. Pixels whose value is 0 indicate “don’t care” pixels in the training image; pixels whose value is 255 indicate “care” pixels in the training image.

  trainParams The training parameters

  diagobj A diagnostic object

  diagFlags Diagnostic flags

  Throws

  ccRSIDefs::BadImageContent
  The Image is featureless.

  ccRSIDefs::BadImage
  image is unbound or has a null window
  mask is unbound
  mask does not have the same window as image
  mask pels have values other than 0 or 255

  ccRSIDefs::NotImplemented
  The image client coordinate system is nonlinear

  ccRSIDefs::BadImageSize
  The image is too small.

  ccTimeout::Expired
  The operation timed out.

untrain

```
void untrain();
```

Untrains this model and releases any saved training data. No effect if the model is not trained.
```

trainParams `ccRSITrainParams trainParams() const;`

Returns **ccRSITrainParams** used to train this model

Throws
ccRSIDefs::NotTrained
The model is not trained.

trainImage `ccPelBuffer_const<c_UInt8> trainImage() const;`

Returns 8-bit image pixels used to train this model

Throws
ccRSIDefs::NotTrained
The model is not trained.
The model was trained using a three-plane image.

trainColorImage
`cc3PlanePelBuffer_const trainColorImage() const;`

Returns three-plane image pixels used to train this model

Throws
ccRSIDefs::NotTrained
The model is not trained.
The model was trained using an 8-bit image.

trainMask `ccPelBuffer_const<c_UInt8> trainMask() const;`

Returns mask image pixels used when this model was trained. If the model was trained without a mask, returns an unbound pel bugger.

Throws
ccRSIDefs::NotTrained
The model is not trained.

isTrained `bool isTrained() const;`

Returns True if model is currently trained, False otherwise

isTrainedMonochrome
`bool isTrainedMonochrome() const;`

Returns True if model is currently trained with a monochrome (8-bit) image, False otherwise

isTrainedColor

```
bool isTrainedColor() const;
```

Returns True if model is currently trained with a color image (three-plane), False otherwise

---

run

```
void run(const ccPelBuffer_const<c_UInt8>& image,
                 const ccRSIRunParams &runParams,
                 ccRSIResultSet &resultSet,
                 ccDiagObject* diagobj=0,
                 c_UInt32 diagFlags=0) const;

void run(const ccPelBuffer_const<c_UInt8>& image,
                 const ccPelBuffer_const<c_UInt8>& mask,
                 const ccRSIRunParams &runParams,
                 ccRSIResultSet &resultSet,
                 ccDiagObject* diagobj=0,
                 c_UInt32 diagFlags=0) const;

void run(const cc3PlanePelBuffer_const& image,
                 const ccRSIRunParams& runParams,
                 ccRSIResultSet& resultSet,
                 ccDiagObject* diagobj=0,
                 c_UInt32 diagFlags=0) const;

void run(const cc3PlanePelBuffer_const& image,
                 const ccPelBuffer_const<c_UInt8>& mask,
                 const ccRSIRunParams& runParams,
                 ccRSIResultSet& resultSet,
                 ccDiagObject* diagobj=0,
                 c_UInt32 diagFlags=0) const;
```

-
```
void run(const ccPelBuffer_const<c_UInt8>& image,
                 const ccRSIRunParams &runParams,
                 ccRSIResultSet &resultSet,
                 ccDiagObject* diagobj=0,
                 c_UInt32 diagFlags=0) const;
```

Searches *image* for this model using the given *runParams*. The *resultSet* is cleared and filled with new results in order of decreasing score.

You can use this function for models that were trained using a mask as well as models that were trained without a mask.

Parameters

image	The 8-bit input image.
runParameters	The search parameters

resultSet The results of the search

diagobj A diagnostic object

diagFlags Diagnostic flags

Throws

ccRSIDefs::NotTrained
The model is not trained.
The model was trained for a three-plane color image.

ccRSIDefs::BadImage
image is unbound.

ccRSIDefs::BadParams
One of the DOF ranges specified in *runParams* (composed by the **startPose()**) exceeds the corresponding DOF in the training parameters of this model.

ccRSIDefs::NotImplemented
The *image* client coordinate system is nonlinear.

ccTimeout::Expired
The function executed longer than the timeout specified. No valid results are produced.

- ```
  void run(const ccPelBuffer_const<c_UInt8>& image,
           const ccPelBuffer_const<c_UInt8>& mask,
           const ccRSIRunParams &runParams,
           ccRSIResultSet &resultSet,
           ccDiagObject* diagobj=0,
           c_UInt32 diagFlags=0) const;
  ```

Searches *image* for this model using *mask* and the given *runParams*. The *resultSet* is cleared and filled with new results in order of decreasing score.

You can use this function for models that were trained using a mask as well as models that were trained without a mask.

Masking is not supported for models trained with compression.

Parameters

image The 8-bit input image.

mask The image mask. Pixels whose value is 0 indicate “don’t care” pixels in the input image; pixels whose value is 255 indicate “care” pixels in the input image.

runParameters The search parameters
  ```

resultSet The results of the search

diagobj A diagnostic object

diagFlags Diagnostic flags

Throws

ccRSIDefs::NotTrained
The model is not trained.
The model was trained for a three-plane color image.

ccRSIDefs::BadImage
image is unbound
mask is unbound
mask does not have the same window as image
mask pels have values other than 0 or 255

ccRSIDefs::BadParams
One of the DOF ranges specified in *runParams* (composed by the **startPose()**) exceeds the corresponding DOF in the training parameters of this model.

ccRSIDefs::NotImplemented
The *image* client coordinate system is nonlinear, or this model was trained with **ccRSITrainParams::compression()** not set to *ccRSIDefs::eCompressionNone*.

ccTimeout::Expired
The function executed longer than the timeout specified. No valid results are produced.

- ```
  void run(const cc3PlanePelBuffer_const& image,
                  const ccRSIRunParams& runParams,
                  ccRSIResultSet& resultSet,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0) const;
  ```

Searches the three-plane *image* for this model using the given *runParams*. The *resultSet* is cleared and filled with new results in order of decreasing score.

You can use this function for models that were trained using a mask as well as models that were trained without a mask.

Parameters

image The three-plane input image.

runParameters The search parameters

resultSet The results of the search
  ```

diagobj A diagnostic object

diagFlags Diagnostic flags

Throws

ccRSIDefs::NotTrained
The model is not trained.
The model was trained for an 8-bit greyscale image.

ccRSIDefs::BadImage
image is unbound

ccRSIDefs::BadParams
One of the DOF ranges specified in *runParams* (composed by the **startPose()**) exceeds the corresponding DOF in the training parameters of this model.

ccRSIDefs::NotImplemented
The *image* client coordinate system is nonlinear.

ccTimeout::Expired
The function executed longer than the timeout specified. No valid results are produced.

- ```
  void run(const cc3PlanePelBuffer_const& image,
                  const ccPelBuffer_const<c_UInt8>& mask,
                  const ccRSIRunParams& runParams,
                  ccRSIResultSet& resultSet,
                  ccDiagObject* diagobj=0,
                  c_UInt32 diagFlags=0) const;
  ```

Searches the three-plane *image* for this model using *mask* and the given *runParams*. The *resultSet* is cleared and filled with new results in order of decreasing score.

You can use this function for models that were trained using a mask as well as models that were trained without a mask.

Masking is not supported for models trained with compression.

Parameters

image The three-plane input image.

mask The image mask. Pixels whose value is 0 indicate “don’t care” pixels in the input image; pixels whose value is 255 indicate “care” pixels in the input image.

runParameters The search parameters

resultSet The results of the search
  ```

diagobj A diagnostic object

diagFlags Diagnostic flags

Throws

ccRSIDefs::NotTrained
The model is not trained.
The model was trained for an 8-bit greyscale image.

ccRSIDefs::BadImage
image is unbound
mask is unbound
mask does not have the same window as image
mask pels have values other than 0 or 255

ccRSIDefs::BadParams
One of the DOF ranges specified in *runParams* (composed by the **startPose()**) exceeds the corresponding DOF in the training parameters of this model.

ccRSIDefs::NotImplemented
The *image* client coordinate system is nonlinear, or this model was trained with **ccRSITrainParams::compression()** not set to *ccRSIDefs::eCompressionNone*.

ccTimeout::Expired
The function executed longer than the timeout specified. No valid results are produced.

ccRSIResult

```
#include <ch_cvl/rsi.h>

class ccRSIResult
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single RSI search result. You should not create a **ccRSIResult** directly.

Constructors/Destructors

ccRSIResult

```
ccRSIResult();
```

Constructs a result object containing no useful data.

Public Member Functions

location

```
cc2Vect location() const;
```

Returns the location of the model origin in the client coordinate system of the runtime image.

The default is (0,0).

pose

```
const cc2Xform& pose() const;
```

Returns a transform from the training time client coordinates (translated to the model origin) to run-time client coordinates. Map **cc2Vect(**0,0**)** through this transform to get the location of the model origin in runtime client coordinates.

The default is **cc2Xform::I**

ccRSIResult

angle

```
ccDegree angle() const;
```

Returns the angle of the located model in the client coordinate system of the runtime image.

The default is 0.

shear

```
ccDegree shear() const;
```

Returns the shear angle of the located model in the client coordinate system of the runtime image.

The default is 0.

xScale

```
double xScale() const;
```

Returns the x-scale of the located model. The default is 1.

yScale

```
double yScale() const;
```

Returns the y-scale of the located model. The default is 1.

scale

```
double scale() const;
```

Returns the scale of the located model. The default is 1.

score

```
double score() const;
```

Returns the score, a number between 0.0 and 1.0. The default is 0.

relativeContrast

```
double relativeContrast() const;
```

Returns the relative contrast of this result. Relative contrast is defined as the scale factor of the standard deviation of the runtime grey levels as compared to the standard deviation of the training time grey levels.

The default is 0.

relativeBrightness

```
double relativeBrightness() const;
```

Returns the relative brightness of this result. Relative brightness is defined as the scale factor of the mean of the runtime grey levels as compared to the mean of the training time grey levels.

The default is 0.

accepted

```
bool accepted() const;
```

Returns True if **score()** is greater than or equal to **ccRSIRunParams::acceptThreshold()**. Otherwise it returns false.

The default is False.

matchRegion

```
ccAffineRectangle matchRegion() const;
```

Returns an affine rectangle that describes the location of the matched model in the runtime image's client coordinates.

The default is **ccAffineRectangle()**.

imageRegion

```
ccPelRect imageRegion() const;
```

Returns the smallest rectangle in runtime image's image coordinates that completely contains the matched model.

default **ccPelRect()**.

draw

```
void draw(ccGraphicList& graphList,
  c_UInt32 drawMode=ccRSIDefs::eDrawStandard,
  const ccCvlString& label=ccCvlString()) const;
```

Appends result graphics for this **ccRSIResult** to the supplied **ccGraphicList**.

Parameters

graphList The graphics list to append the graphics to.

drawMode The drawing mode to use. *drawMode* must be composed by ORing together one or more of the following values:

ccRSIDefs::eDrawOrigin
ccRSIDefs::eDrawLabel
ccRSIDefs::eDrawBoundingBox
ccRSIDefs::eDrawStandard

label	If you include *ccRSIDefs::eDrawLabel* in *drawMode*, then the contents of *label* are used to label the origin of this result.

Notes

Graphics are drawn in **ccColor::greenColor()** if **accepted()** is True for this result, **ccColor::redColor()** if it is not.

ccRSIResultSet

```
#include <ch_cvl/rsi.h>

class ccRSIResultSet
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains all of the **ccRSIResult** objects returned by an RSI search.

Constructors/Destructors

ccRSIResultSet

```
ccRSIResultSet();

~ccRSIResultSet();

ccRSIResultSet(const ccRSIResultSet&);
```

- `ccRSIResultSet();`

  Creates a **ccRSIResultSet** with no results. **numFound()** and **time()** both return 0 when called on a default-constructed **ccRSIResultSet**.

Public Member Functions

numFound

```
c_Int32 numFound() const;
```

Returns the number of **ccRSIResult**s in this **ccRSIResultSet**.

results

```
const cmStd vector<ccRSIResult>& results() const;
```

Returns a vector of **ccRSIResult**s that contains all the results from this search.

ccRSIResultSet

ccRSIRunParams

```
#include <ch_cvl/rsi.h>

class ccRSIRunParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters used to control an RSI search.

This class lets you specify the runtime parameters used to control an RSI search. These parameters include degrees of freedom (DOF) for four *zones*: angle, uniform scale, x-scale and y-scale. You specify which zones are enabled and the range in each zone the tool can use. For zones not enabled, the tool uses a nominal value instead of a range of values.

If the **ccRSIModel** was trained with **ccRSITrainParams::mode()** set to *ccRSIDefs::eGenerateTemplates*, then to use a zone during a search, the tool must have been trained to use that zone. You don't have to enable a zone that you trained for, but you cannot enable a zone that was not trained.

The runtime zones are relative to the start pose. For example, if the start pose is rotated by 30 degrees, then enabling the rotation DOF by -10 degrees to +10 degrees would mean that the tool would search from +20 degrees to +40 degrees. If the start pose has a uniform scale of 1.05, enabling the uniform scale DOF from 0.9 to 1.1 means that the tool would search from 0.945 (0.9 * 1.05) to 1.155 (0.9 *

The following table summarizes these four DOF zones, their defaults, and their ranges. All scale values must be greater than 0.

DOF	Units	Nom.	Default		Range	Zone Enable	Restrictions
			Low	High			
ccRSIDefs::eAngle	degrees	0	-45	+45	any	False	none
ccRSIDefs::eShear	degrees	0	0	0	any	False	none
ccRSIDefs::eUniformScale	multiplier	1	0.8	1.2	> 0	False	zoneLow <= zoneHigh

Table 1. Zone defaults and ranges

DOF	Units	Nom.	Default Low	Default High	Range	Zone Enable	Restrictions
ccRSIDefs::eXScale	multiplier	1	0.8	1.2	> 0	False	zoneLow <= zoneHigh
ccRSIDefs::eYScale	multiplier	1	0.8	1.2	> 0	False	zoneLow <= zoneHigh

Table 1. Zone defaults and ranges

The *angle* zone is always searched from zoneLow to zoneHigh, so setting zoneLow to 0 and zoneHigh to 180 searches one half of the 360 degree angle range. Setting zoneHigh to 0 and zoneLow to 180 searches the other half.

x-scale and *y-scale* only scale in the given dimension. *Uniform scale* performs equal scale in both the x- and y-dimensions. Choose either uniform scale or a combination of x-scale and y-scale. Do not use both uniform scale and x- and y-scale.

Uniform scale is specified as a multiplier of the trained model size. For example, a uniform scale of 1.25 implies that the run-time (client coordinate) size of the model will be 25% larger in both dimensions than the trained model.

Constructors/Destructors

ccRSIRunParams

```
ccRSIRunParams();
~ccRSIRunParams();
ccRSIRunParams(const ccRSIRunParams&);
```

Constructs a **ccRSIRunParams** object using the default values.

Public Member Functions

acceptThreshold

```
double acceptThreshold() const;
void acceptThreshold(double a);
```

- `double acceptThreshold() const;`

  Returns the acceptance threshold.

- `void acceptThreshold(double a);`

  Sets a new acceptance threshold. A result is considered accepted if its score is greater than or equal to this value.

  Parameters

  a The threshold. Must be in the range 0.0 through 1.0. The default value is 0.5.

  Throws

  ccRSIDefs::BadParams

  a < 0 or > 1.0

confusionThreshold

```
double confusionThreshold() const;
void confusionThreshold(double c);
```

- `double confusionThreshold() const;`

  Returns the current confusion threshold.

- `void confusionThreshold(double c);`

  Sets a new confusion threshold.

  Parameters

  c The new confusion threshold.

  Throws

  ccRSIDefs::BadParams

  c < 0 or > 1.0

  Notes

  If the confusion threshold is < **acceptThreshold(), ccRSIModel::run()** will throw *ccRSIDefs::BadParams.*

relativeContrastRange

```
ccRange relativeContrastRange() const;

void relativeContrastRange(
                    const ccRange &relativeContrastRange);
```

- `ccRange relativeContrastRange() const;`

  Returns the current relative contrast range.

- `void relativeContrastRange(`
  `const ccRange &relativeContrastRange);`

  Sets a new relative contrast range, where contrast is defined as the relative scale factor of the standard deviation of the runtime grey levels as compared to the standard deviation of the training time grey levels. Candidate results with lower contrast are removed and never reported as found.

  Parameters

  relativeContrastRange
  The new relative contrast range. The default is **ccRange(**0,HUGE_VAL**)**.

  Throws

  ccRSIDefs::BadParams
  If *relativeContrastRange* has a negative start value, or if *relativeContrastRange* is empty.

  Notes

  Setting *relativeContrastRange* to **ccRange(**0,HUGE_VAL**)** effectively disables the constraint.

relativeBrightnessRange

```
ccRange relativeBrightnessRange() const;

void relativeBrightnessRange(
                 const ccRange &relativeBrightnessRange);
```

- `ccRange relativeBrightnessRange() const;`

  Returns the current relative brightness range.

- `void relativeBrightnessRange(const ccRange &relativeBrightnessRange);`

Sets a new relative brightness range, where brightness is defined as the relative scale factor of the mean of the runtime grey levels as compared to the mean of the training time grey levels. Candidate results with lower or higher brightness are removed and never reported as found.

Parameters

relativeBrightnessRange
The new relative brightness range. The default is **ccRange(**0,HUGE_VAL**)**.

Throws

ccRSIDefs::BadParams
If *relativeBrightnessRange* has a negative start value, or if *relativeBrightnessRange* is empty.

Notes

Setting *relativeBrightnessRange* to **ccRange(**0,HUGE_VAL**)** effectively disables the constraint.

ignorePolarity

```
bool ignorePolarity() const;
void ignorePolarity(bool t);
```

- `bool ignorePolarity() const;`

Returns the current ignore polarity parameter.

- `void ignorePolarity(bool t);`

Sets the ignore polarity parameter. When the ignore polarity parameter is True, the exact grey level inverse of the model can be found.

Parameters

t The new setting. The default is False.

numToFind

```
c_Int32 numToFind() const;
void numToFind(c_Int32 n);
```

- `c_Int32 numToFind() const;`

Returns the number of results to look for.

- `void numToFind(c_Int32 n);`

  Sets the number of results to look for. The **ccRSIModel::run()** function may return fewer results, but it will never return more results than you specify here.

  Parameters

  n The number of results to look for. The default is 1.

  Throws

  ccRSIDefs::BadParams
  If *n* <= 0

startPose

```
cc2XformLinear startPose() const;

void startPose(const cc2XformLinear &startPose);
```

- `cc2XformLinear startPose() const;`

  Returns the current start pose.

- `void startPose(const cc2XformLinear &startPose);`

  Sets the start pose. This start pose maps the model from train-time coordinates into runtime coordinates. If the start pose is exactly correct, then the found result will have exactly the same pose.

  Parameters

  startPose The new start pose. The default is **cc2XformLinear()**.

  Throws

  ccRSIDefs::BadParams
  If *startPose* is singular.

translationUncertainty

```
ccRect translationUncertainty() const;

void translationUncertainty(
                  const ccRect& translationUncertainty);
```

- `ccRect translationUncertainty() const;`

  Returns a rectangle that defines the translation uncertainty bounds.

- `void translationUncertainty(`
  `const ccRect& translationUncertainty);`

  Sets a rectangle that defines the translation uncertainty bounds. The translation uncertainty is specified in client coordinate units and is aligned to the client coordinate axes. The rectangle may be (0, 0, 0, 0) to indicate a point search. Otherwise, the rectangle must contain the point (0, 0).

  Parameters

  translationUncertainty

  The new translation uncertainty. The default is **ccRect(**-HUGE_VAL,-HUGE_VAL,HUGE_VAL,HUGE_VAL**)**;

  Throws

  ccRSIDefs::BadParams

  If *translationUncertainty* does not contain the point (0, 0) and is not the rect (0, 0, 0, 0),
  or if *translationUncertainty* has HUGE_VAL or -HUGE_VAL for any corner or size except for the fully infinite rect (-HUGE_VAL,-HUGE_VAL,HUGE_VAL,HUGE_VAL).

---

xyOverlap

`double xyOverlap() const;`

`void xyOverlap(double overlapThresh);`

- `double xyOverlap() const;`

  Returns the current x/y overlap threshold.

- `void xyOverlap(double overlapThresh);`

  Sets a new x/y overlap threshold required to consider two instances to be the same result. Two results are considered overlapping if the overlap value (approximately the ratio of overlap area to the total area of the model) is greater than *overlapThresh*.

  Meaningful values range from 0.0 to 1.0. When *overlapThresh* is 1.0, no results overlap (overlapping is disabled). When *overlapThresh* is 0.0, results with non-null intersections overlap.

  Parameters

  overlapThresh The new x/y overlap threshold. The default value is 0.8.

  Throws

  ccRSIDefs::BadParams

  If *overlapThresh* is outside the range 0.0 through 1.0.

zoneEnable

```
c_UInt32 zoneEnable() const;
void zoneEnable(c_UInt32 enable);
```

- `c_UInt32 zoneEnable() const;`

  Returns the current zone settings.

- `void zoneEnable(c_UInt32 enable);`

  Sets which degrees-of-freedom (DOF) zones to consider when searching. The non-zero bits in enable correspond to the values of the **ccRSIDefs::DOF** constants.

  If a zone is enabled for a particular DOF, **ccRSIModel::run()** searches between **zoneLow()** and **zoneHigh()** for that DOF. Otherwise it uses the **nominal()** value for that DOF.

  If a DOF zone is enabled, **run()** results may fall slightly outside **zoneLow()** and **zoneHigh()**. If no DOF zone enabled, the result that **run()** produces will always be **nominal()** for that DOF.

  Parameters

  enable — The enabled zones.
  The default is zero: no DOF zones enabled.

  Throws

  ccRSIDefs::BadParams

  enable is not zero or a combination created by a bitwise-OR of the values from **ccRSIDefs::DOF**.

nominal

```
double nominal(ccRSIDefs::DOF dof) const;
void nominal(ccRSIDefs::DOF dof, double val);
```

- `double nominal(ccRSIDefs::DOF dof) const;`

  Returns the nominal value for the specified DOF.

  Parameters

  dof — The specified degree of freedom.

- `void nominal(ccRSIDefs::DOF dof, double val);`

  Sets a nominal value for the specified DOF.

  The nominal value for a given degree of freedom (DOF) is used only if it is not enabled by **zoneEnable()** The **zoneLow()** and **zoneHigh()** values are ignored when the nominal value is used.:

DOF	Default Nominal Value
ccRSIDefs::eAngle	0.0
ccRSIDefs::eShear	0.0
ccRSIDefs::eUniformScale	1.0
ccRSIDefs::eXScale	1.0
ccRSIDefs::eYScale	1.0

  Parameters

  dof The specified degree of freedom.

  val The value to set.

  Throws

  ccRSIDefs::BadParams

  dof is not in ccRSIDefs::DOF
  val <= 0 and *dof* is **ccRSIDefs::eUniformScale, ccRSIDefs::eXScale,** or **ccRSIDefs::eYScale**

zoneLow `double zoneLow (ccRSIDefs::DOF dof) const;`

Returns the low value for the specified degree of freedom (DOF). **ccRSIModel::run()** searches between **zoneLow()** and **zoneHigh()** for a specified DOF only when it is enabled with **zoneEnable()**.

dof The specified degree of freedom.

zoneHigh `double zoneHigh(ccRSIDefs::DOF dof) const;`

Returns the high value for the specified degree of freedom (DOF). **ccRSIModel::run()** searches between **zoneLow()** and **zoneHigh()** for a specified DOF only when it is enabled with **zoneEnable()**

Parameters

dof The specified degree of freedom.

zone

```
void zone(ccRSIDefs::DOF dof, double low, double high);
```

Sets the low and high values for the specified degree of freedom (DOF). **ccRSIModel::run()** searches between **zoneLow()** and **zoneHigh()** for a specified DOF only when it is enabled with **zoneEnable()**

Notes

The span of each degree of freedom's runtime zone must be smaller than or equal to the span of the corresponding training zone or a runtime exception will be thrown.

Parameters

dof The specified degree of freedom.

low The low value.

high The high value.

Throws

ccRSIDefs::BadParams

dof is not in **ccRSIDefs::DOF**
low and *high* do not meet the restrictions listed in Table 1.

zoneOverlap

```
double zoneOverlap(ccRSIDefs::DOF dof) const;
void zoneOverlap(ccRSIDefs::DOF dof, double overlapThresh);
```

- `double zoneOverlap(ccRSIDefs::DOF dof) const;`

Return the overlap threshold for the specified degree of freedom.

Parameters

dof The specified degree of freedom.

- `void zoneOverlap(ccRSIDefs::DOF dof, double overlapThresh);`

  Sets the zone overlap threshold for the specified degree of freedom. The meaning of overlap is different for angles and for scales.

  For **ccRSIDefs::eAngle**, two results are considered overlapping if the absolute difference between their angles is less than or equal to *overlapThresh*. This means that for **ccRSIDefs::eAngle**, *overlapThresh* must be greater than or equal to zero.

  For the other degrees of freedom overlap is specified as a ratio of the result with the greater scale to the result with the smaller scale. If the ratio of the two results is less than or equal to *overlapThresh*, the results are considered overlapping. This means that overlapThresh must be greater than 1.0.

  overlapThresh must be >= 1.0. For example, if *dof* is one of the scale degrees of freedom, and *overlapThresh* = 1.2, then two results, one with a scale of 1.25 and one with a scale of 2.5, would not overlap in scale since 2.5/1.25 is greater than 1.2.

  Parameters

dof	The specified degree of freedom.
overlapThresh	The overlap threshold. For angles, the default is 360.0. FOr scales, the default is 1.4.

  Throws

  ccRSIDefs::BadParams
  overlapThresh is not valid for the specified *dof*.

method

```
ccRSIDefs::Method method() const;

void method(ccRSIDefs::Method method);
```

- `ccRSIDefs::Method method() const;`

  Returns the method used to determine the sub-pixel position. The returned value is one of the following:

  ccRSIDefs::eStandard
  ccRSIDefs::eHighAccuracy

- `void method(ccRSIDefs::Method method);`

  Sets the method used to determine the sub-pixel position. The default value is *ccRSIDefs::eStandard*.

Parameters

method The method to use. *method* must be one of the following values:

ccRSIDefs::eStandard
ccRSIDefs::eHighAccuracy

Throws

BadParams

method is neither *eStandard* nor *eHighAccuracy*.

forceUncompressedForScore

```
bool forceUncompressedForScore() const;

void forceUncompressedForScore(bool f);
```

- `bool forceUncompressedForScore() const;`

  Returns true if RSI search uses an uncompressed version of the model to compute the score, false otherwise.

- `void forceUncompressedForScore(bool f);`

  Sets whether or not to use an uncompressed verison of the trained model to compute the score.

  Parameters

  f The new setting. The default is True.

  Notes

  This setting has no effect if the model was trained with **ccRSIRunParams::compression()** set to *ccRSIDefs::eCompressionNone*.

ccRSITrainParams

```
#include <ch_cvl/rsi.h>

class ccRSITrainParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

When you train the RSI Search tool you specify parameters the tool uses during searches. These parameters include degrees of freedom (DOF) for four *zones*: angle, uniform scale, x-scale and y-scale. You specify which zones are enabled and the range in each zone the tool can use. For zones not enabled, the tool uses a nominal value instead of a range of values.

The following table summarizes these four DOF zones, their defaults, and their ranges. All scale values must be greater than 0.

DOF	Units	Nom.	Default Low	Default High	Range	Zone Enable	Restrictions
ccRSIDefs::eAngle	degrees	0	-45	+45	any	False	none
ccRSIDefs::eUniformScale	multiplier	1	0.8	1.2	> 0	False	zoneLow <= zoneHigh
ccRSIDefs::eXScale	multiplier	1	0.8	1.2	> 0	False	zoneLow <= zoneHigh
ccRSIDefs::eYScale	multiplier	1	0.8	1.2	> 0	False	zoneLow <= zoneHigh
ccRSIDefs::eShear	degrees	0	0	0	any	False	zoneLow <= zoneHigh

Table 1. Zone defaults and ranges

The *angle* zone is always searched from zoneLow to zoneHigh, so setting zoneLow to 0 and zoneHigh to 180 searches one half of the 360 degree angle range. Setting zoneHigh to 0 and zoneLow to 180 searches the other half.

x-scale and *y-scale* only scale in the given dimension. *Uniform scale* performs equal scale in both the x- and y-dimensions.

Uniform scale is specified as a multiplier of the trained model size. For example, a uniform scale of 1.25 implies that the run-time (client coordinate) size of the model will be 25% larger in both dimensions than the trained model.

Constructors/Destructors

ccRSITrainParams

```
ccRSITrainParams();
```

Constructs an RSI Search tool training parameters object using the default values.

Public Member Functions

mode

```
ccRSIDefs::Mode mode() const;

void mode(ccRSIDefs::Mode mode);
```

mode() controls whether RSI Search creates template images at training time or at run time.

When **mode()** is set to *ccRSIDefs::eGenerateTemplates*, RSI Search creates, at training time, all of the template images required to cover the search space that is specified by the **zone()**, **nominal(), zoneEnable(), zoneLow()**, and **zoneHigh()** members of this class.

When **mode()** is set to *ccRSIDefs::eUnrestrictedDOFs*, RSI Search does not create any template images at training time. Instead, the tool automatically generates template images at run time, based on the search space specified for the particular search.

In general, *ccRSIDefs::eGenerateTemplates* provides for faster operation at run time, but specifying this mode creates the requirement that the search space specified for training must enclose the search space for each run-time search.

- `ccRSIDefs::Mode mode() const;`

  Returns the current mode. The returned value is one of the following values:

  ccRSIDefs::eGenerateTemplates
  ccRSIDefs::eUnrestrictedDOFs

- `void mode(ccRSIDefs::Mode mode);`

  Sets the mode. The default value is *ccRSIDefs::eGenerateTemplates*.

mode	The mode to set. *mode* must be one of the following values: *ccRSIDefs::eGenerateTemplates* *ccRSIDefs::eUnrestrictedDOFs*

Throws

ccRSIDefs::BadParams

mode is neither *ccRSIDefs::eGenerateTemplates nor ccRSIDefs::eUnrestrictedDOFs*.

combineDOF

```
ccRSIDefs::CombineDOF combineDOF() const;

void combineDOF(ccRSIDefs::CombineDOF c);
```

This value of **combineDOF()** is used by RSI Search to interpret the meaning of **ccRSIModel::trainClientFromModel()**.

When **combineDOF()** is set to *ccRSIDefs::eCombineDOFTrainClientFromModel*, RSI Search interprets the values specified for the **zone()**, **nominal(), zoneLow()**, and **zoneHigh()** members as having been combined with the values for the corresponding degrees of freedom found in **ccRSIModel::trainClientFromModel()**.

When **combineDOF()** is set to *ccRSIDefs::eCombineDOFIdentity*, RSI Search interprets the values specified for the **zone()**, **nominal(), zoneLow()**, and **zoneHigh()** members without regard to the values for the corresponding degrees of freedom found in **ccRSIModel::trainClientFromModel()**.

The value specified for this member is only used when **mode()** is set to *ccRSIDefs::eGenerateTemplates* and **ccRSIModel::trainClientFromModel()** is set to a non-identity transform.

- `ccRSIDefs::CombineDOF combineDOF() const;`

  Returns the current DOF combination mode. The returned value is one of the following:

  ccRSIDefs::eCombineDOFTrainClientFromModel
  ccRSIDefs::eCombineDOFIdentity

- `void combineDOF(ccRSIDefs::CombineDOF c);`

  Sets the current DOF combination mode.

  Parameters

  c — The combination mode to set. *c* must be one of the following values:

  ccRSIDefs::eCombineDOFTrainClientFromModel
  ccRSIDefs::eCombineDOFIdentity

Throws

ccRSIDefs::BadParams

c is neither *ccRSIDefs::eCombineDOFTrainClientFromModel nor ccRSIDefs::eCombineDOFIdentity*.

zoneEnable

```
c_UInt32 zoneEnable() const;

void zoneEnable(c_UInt32 enable);
```

The value of this member is ignored if **mode()** is set to *ccRSIDefs::eGenerateTemplates*

- `c_UInt32 zoneEnable() const;`

  Returns the current zone settings.

- `void zoneEnable(c_UInt32 enable);`

  Sets which degrees-of-freedom (DOF) zones to consider when training. The non-zero bits in enable correspond to the values of the **ccRSIDefs**::**DOF** constants.

  If a zone is enabled for a particular DOF, **ccRSIModel::train()** trains between **zoneLow()** and **zoneHigh()** for that DOF. Otherwise it trains the **nominal()** value for that DOF.

  Parameters

  enable — The enabled zones. The default is zero: no DOF zones enabled.

  Throws

  ccRSIDefs::BadParams

  enable is not zero or a combination created by a bitwise-OR of the values from **ccRSIDefs::DOF**.

  Notes

  The value of this member is ignored if **mode()** is set to *ccRSIDefs::eUnrestrictedDOFs.*

nominal

```
double nominal(ccRSIDefs::DOF dof) const;

void nominal(ccRSIDefs::DOF dof, double val);
```

- `double nominal(ccRSIDefs::DOF dof) const;`

  Returns the nominal value for the specified DOF.

Parameters

dof The degree of freedom.

- `void nominal(ccRSIDefs::DOF dof, double val);`

The nominal value for a given degree of freedom (DOF) is used only if it is not enabled by **zoneEnable()** The **zoneLow()** and **zoneHigh()** values are ignored when the nominal value is used.:

DOF	Default Nominal Value
ccRSIDefs::eAngle	0.0
ccRSIDefs::eUniformScale	1.0
ccRSIDefs::eXScale	1.0
ccRSIDefs::eYScale	1.0
ccRSIDefs::eShear	0.0

Parameters

dof The specified degree of freedom.

val The value to set.

Throws

ccRSIDefs::BadParams

dof is not in ccRSIDefs::DOF
val <= 0 and *dof* is *ccRSIDefs::eUniformScale*, *ccRSIDefs::eXScale*, *ccRSIDefs::eYScale* or *ccRSIDefs::eShear*.

Notes

The value of this member is ignored if **mode()** is set to *ccRSIDefs::eUnrestrictedDOFs*.

zoneLow `double zoneLow (ccRSIDefs::DOF dof) const;`

Returns the low value for the specified degree of freedom (DOF). **ccRSIModel::train()** trains between **zoneLow()** and **zoneHigh()** for a specified DOF only when it is enabled with **zoneEnable()**.

dof The specified degree of freedom.

Notes

The value of this member is ignored if **mode()** is set to *ccRSIDefs::eUnrestrictedDOFs*.

zoneHigh

```
double zoneHigh(ccRSIDefs::DOF dof) const;
```

Returns the high value for the specified degree of freedom (DOF). **ccRSIModel::train()** trains between **zoneLow()** and **zoneHigh()** for a specified DOF only when it is enabled with **zoneEnable()**.

dof The specified degree of freedom.

Notes

The value of this member is ignored if **mode()** is set to *ccRSIDefs::eUnrestrictedDOFs*.

zone

```
void zone(ccRSIDefs::DOF dof, double low, double high);
```

Sets the low and high values for the specified degree of freedom (DOF). **ccRSIModel::train()** trains between **zoneLow()** and **zoneHigh()** for a specified DOF only when it is enabled with **zoneEnable()**

Notes

The span of each degree of freedom's runtime zone must be smaller than or equal to the span of the corresponding training zone or a runtime exception will be thrown.

Parameters

dof The specified degree of freedom.

low The low value.

high The high value.

Throws

ccRSIDefs::BadParams

dof is not in **ccRSIDefs::DOF**

low and *high* do not meet the restrictions listed in Table 1.

Notes

The value of this member is ignored if **mode()** is set to *ccRSIDefs::eUnrestrictedDOFs*.

autoSelectGrainLimits

```
bool autoSelectGrainLimits() const;

void autoSelectGrainLimits(bool b);
```

Returns whether granularity limits are set automatically.

Notes

Enabling automatic selection generally doubles training time.

- `bool autoSelectGrainLimits() const;`

  Returns the current automatic selection setting.

- `void autoSelectGrainLimits(bool b);`

  Allows the automatic selection of granularity limits at training time. The default is True.

  Parameters

  b The new automatic selection flag.

coarseGrainLimit

`float coarseGrainLimit () const;`

Returns the current coarse granularity limit setting. Ignored if **autoSelectGrainLimits()** is True.

fineGrainLimit

`float fineGrainLimit () const;`

Returns the current fine granularity limit setting. Ignored if **autoSelectGrainLimits()** is True.

grainLimits

`void grainLimits(float coarse, float fine);`

Sets the coarse and fine granularity limits.

Notes

If **autoSelectGrainLimits()** is true, these limit values are ignored.

Parameters

coarse The new coarse granularity limit. Default is 4.0.

fine The new fine granularity limit. Default is 1.0.

Throws

ccRSIDefs::BadParams

If *coarse* < *fine,*
or if coarse < 1,
or if fine < 1

granularityGenerator

```
ccRSIDefs::GranularityGenerator
   granularityGenerator() const;

void granularityGenerator(
   ccRSIDefs::GranularityGenerator g);
```

Sets or gets the method used to generate the coarse granularity model. The default value, *ccRSIDefs::eGranularityGeneratorRobust*, provides the best results in most cases, and is guaranteed to find all valid model instances. If you specify *eGranularityGeneratorFast*, it can improve run-time speed but when used with models containing primarily fine features, it can fail to find some model instances.

The default value is *ccRSIDefs::eGranularityGeneratorRobust*.

- ```
  ccRSIDefs::GranularityGenerator
     granularityGenerator() const;
  ```

  Returns the current coarse granularity generator. The returned value is one of the following:

  ccRSIDefs::eGranularityGeneratorRobust
  ccRSIDefs::eGranularityGeneratorFast

- ```
  void granularityGenerator(
     ccRSIDefs::GranularityGenerator g);
  ```

  Sets the coarse granularity generator.

  Parameters

  g — The generator to use. *g* must be one of the following values:

  cRSIDefs::eGranularityGeneratorRobust
  ccRSIDefs::eGranularityGeneratorFast

compression

```
ccRSIDefs::Compression compression() const;
void compression(ccRSIDefs::Compression c);
```

Sets or gets the compression mode for this model. Using compression reduces the amount of data required to represent the trained model and can improve run-time execution speed. The use of *ccRSIDefs::eCompressionLossless* is not recommended. You can specify the degree of lossy compression, when using *ccRSIDefs::eCompressionLossy*, using **lossyCompressionQuality()**.

The default value is *ccRSIDefs::eCompressionNone*.

- `ccRSIDefs::Compression compression() const;`

  Returns the current compression mode. The returned value is one of the following:

  ccRSIDefs::eCompressionNone
  ccRSIDefs::eCompressionLossy
  ccRSIDefs::eCompressionLossless

- `void compression(ccRSIDefs::Compression c);`

  Sets the compression mode.

  Parameters

  c — The compression mode to set. *c* must be one of the following values:

  ccRSIDefs::eCompressionNone
  ccRSIDefs::eCompressionLossy
  ccRSIDefs::eCompressionLossless

lossyCompressionQuality

```
double lossyCompressionQuality() const;

void lossyCompressionQuality(double q);
```

Sets or gets the degree of compression that is applied. This only applies when **compression()** is *ccRSIDefs::eCompressionLoss.* The compression quality must be greater than zero and less than or equal to 1.0. Larger values indicate higher quality and lower compression.

In general, higher values produce slower, more robust searches, while lower values produce faster, less robust searches.

The default value is 0.95.

- `double lossyCompressionQuality() const;`

  Returns the current compression quality. The returned value is greater than 0 and less than or equal to 1.0.

- `void lossyCompressionQuality(double q);`

  Sets the compression quality.

  Parameters

q	The compression quality. The supplied value must be greater than 0 and less than or equal to 1.0.

ccSampleConvolveParams

```
#include <ch_cvl/smplconv.h>

class ccSampleConvolveParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class defines parameter objects used by the **cfSampleConvolve()** global function.

Constructors/Destructors

ccSampleConvolveParams

```
ccSampleConvolveParams();
```

Constructs object using the default values. The default values are the following:

sample_ = (1.0, 1.0)

(The internally stored sample value.)

kernelX_ = (empty)

(The internally stored calculated or provided 1D kernel in x.)

kernelY_ = (empty)

(The internally stored calculated or provided 1D kernel in y.)

canUsePelRoot_ = true

(The internally stored flag to indicate whether it is okay to use pels outside the src **ccPelBuffer**'s rect in its pelroot.)

Notes

The default copy constructor, assignment operator, and destructor are used.

Public Member Functions

sample

```
ccDPair sample() const;
void sample(const ccDPair &samp);
void sample(double samp);
```

- `ccDPair sample() const;`

  Gets the sampling rates in x and y.

- `void sample(const ccDPair &samp);`

  Sets the sampling rates in x and y.

  Parameters

  samp The sampling rates to be set in x and y.

  The default is (1.0, 1.0).

  Throws

  ccSampleConvolveDefs::BadParams

  samp.x() <= 0 or **samp.y()** <= 0

- `void sample(double samp);`

  Sets the sampling rates in x and y. This overload sets sampling rates in both directions to the same value.

  For example, to specify sampling every third pixel, set sample to (3,3).

  Parameters

  samp The sampling rates to be set in x and y.

  The default is (1.0, 1.0).

  Notes

  Non-integer sampling is fully supported.

  Non-integer sampling is typically slightly slower than sampling of FLOOR (sample) and significantly slower than sampling of CEIL(sample).

kernelX

```
const cmStd vector<c_Int16> &kernelX() const;
void kernelX(const cmStd vector<c_Int16> &kX);
```

- `const cmStd vector<c_Int16> &kernelX() const;`

  Gets the kernel used for separable X convolution.

- `void kernelX(const cmStd vector<c_Int16> &kX);`

  Sets the kernel used for separable X convolution.

  The setter makes a copy of the provided vector, while the getter returns a const reference to the stored vector.

  Parameters

  kX The kernel to be set for separable X convolution.

  Throws

  ccSampleConvolveDefs::NotImplemented
  The kernel is larger than 128 elements or the kernel size is an even number,
  or the sum of all positive elements of the kernel is greater than 128, or the sum of all negative elements of the kernel is less than -128.

  Notes

  Default parameters cannot be used for convolve/sampling. A valid kernel must be set, either directly or by using **setGaussSample()** or **setGaussSmoothing()**.

kernelY

```
const cmStd vector<c_Int16> &kernelY() const;
void kernelY(const cmStd vector<c_Int16> &kY);
```

- `const cmStd vector<c_Int16> &kernelY() const;`

  Gets the kernel used for separable Y convolution.

- `void kernelY(const cmStd vector<c_Int16> &kY);`

  Sets the kernel used for separable Y convolution.

  The setter makes a copy of the provided vector, while the getter returns a const reference to the stored vector.

Parameters

kY The kernel to be set for separable Y convolution.

Throws

ccSampleConvolveDefs::NotImplemented
The kernel is larger than 128 elements or the kernel size is an even number,
or the sum of all positive elements of the kernel is greater than 128, or the sum of all negative elements of the kernel is less than -128.

Notes

Default parameters cannot be used for convolve/sampling. A valid kernel must be set, either directly or by using **setGaussSample()** or **setGaussSmoothing()**.

setGaussSample

```
void setGaussSample(const ccDPair &samp);

void setGaussSample(double samp);
```

- `void setGaussSample(const ccDPair &samp);`

Quickly sets up sample and kernel values for Gaussian smoothing in a single function call. The sample value is set to the provided value (*samp*) as in the **sample()** setters, and 2D Gaussian kernel is defined with sigma = 0.5 * **floor(***samp***)**

Parameters

samp The sample value to be set for Gaussian smoothing.

Notes

These parameters emphasize speed of operation and ease of use. Other kinds of convolution can be defined explicitly using the **sample()**, **kernelX()**, and **kernelY()** setters.

Throws

ccSampleConvolveDefs::NotImplemented
Any *samp* value is greater than 64.

ccSampleConvolveDefs::BadParams
Any *samp* value is less than or equal to 0.

- `void setGaussSample(double samp);`

Quickly sets up sample and kernel values for Gaussian smoothing in a single function call. The sample value is set to the provided value (*samp*) as in the **sample()** setters, and 2D Gaussian kernel is defined with sigma = 0.5 * **floor(***samp***)**

Parameters

samp The sample value to be set for Gaussian smoothing.

Notes

This parameter emphasizes speed of operation and ease of use. Other kinds of convolution can be defined explicitly using the **sample()**, **kernelX()**, and **kernelY()** setters.

Throws

ccSampleConvolveDefs::NotImplemented
Any *samp* value is greater than 64.

ccSampleConvolveDefs::BadParams
Any *samp* value is less than or equal to 0.

setGaussSmoothing

```
void setGaussSmoothing(const ccDPair &sigma);

void setGaussSmoothing(double sigma);
```

- `void setGaussSmoothing(const ccDPair &sigma);`

Quickly sets up kernel values for Gaussian smoothing in a single function call. A 2D Gaussian kernel is defined with the given sigma value(s). The sample rate is left unchanged.

Parameters

sigma The kernel values to be set for Gaussian smoothing.

Notes

These parameters emphasize speed of operation and ease of use. Other kinds of convolution can be defined explicitly using the **kernelX()** and **kernelY()** setters.

Throws

ccSampleConvolveDefs::NotImplemented
Any sigma value is greater than 32.

ccSampleConvolveDefs::BadParams
Any sigma value is less than 0.

- `void setGaussSmoothing(double sigma);`

Quickly sets up kernel values for Gaussian smoothing in a single function call. A 2D Gaussian kernel is defined with the given sigma value. The sample rate is left unchanged.

Parameters

sigma The kernel value to be set for Gaussian smoothing.

Notes

This parameter emphasizes speed of operation and ease of use. Other kinds of convolution can be defined explicitly using the **kernelX()** and **kernelY()** setters.

Throws

ccSampleConvolveDefs::NotImplemented
The *sigma* value is greater than 32.

ccSampleConvolveDefs::BadParams
The *sigma* value is less than 0.

canUsePelRoot

```
bool canUsePelRoot () const;

void canUsePelRoot (bool canUseRoot);
```

- `bool canUsePelRoot () const;`

  Gets whether or not it is okay to use pels outside the src **ccPelBuffer**'s rect in its pelroot.

- `void canUsePelRoot (bool canUseRoot);`

  Sets whether or not it is okay to use pels outside the src **ccPelBuffer**'s rect in its pelroot.

  Parameters

  canUseRoot Whether or not it is okay to use pels outside the src **ccPelBuffer**'s rect in its pelroot.

  The default is true.

computeDestRect

```
void computeDestRect(
  const ccRect &srcRect,
  ccRect &destRect) const;
```

Computes the maximum destination region that could result from the given source region and parameters.

Parameters

srcRect The source region.

destRect The destination region.

Notes

If **cfSampleConvolve()** is called with an unbound destination, the result rect is exactly the return result of this function. Otherwise, the result of the tool will be to fill in the GCR of the input destination window and the window that results from this function.

The relationship of *srcRect* and *destRect* depends both on the sampling factor and (in general) the sub-pixel offset.

computeMaxSrcRect

```
void computeMaxSrcRect(
  const ccRect &srcRect,
  const ccRect &destRect,
  ccRect &maxSrcRect) const;
```

Computes the maximum source region that would be referenced by this operation.

Parameters

srcRect The source region.

destRect The destination region.

maxSrcRect The maximum source region.

Notes

The rectangle returned by this function contains exactly the src input pels that are "referenced" if the same parameters are sent to **cfSampleConvolve()**. In particular, if **cfSampleConvolve()** is called with a src image that contains *maxSrcRect*, or (if **canUsePelRoot()** is true) with a src image root that contains *maxSrcRect*, then no input pels are artificially produced by reflection.

ccSampleConvolveParams

ccSampleParams

```
#include <ch_cvl/shapbase.h>

class ccShape::ccSampleParams;
```

This class specifies shape sampling parameters. An instance of this class is passed to **ccShape::sample()** to describe how the sampling should be done. This is a nested class defined inside of **ccShape**.

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

Constructors/Destructors

ccSampleParams

```
ccSampleParams(const double &tolerance = 0.0,
  const double &spacing = 0.0,
  c_Int32 maxPoints = 2,
  bool computeTangents = false,
  bool duplicateCorners = true);
```

Constructs a **ccSampleParams** with the given parameters. The default parameter values construct a **ccSampleParams** set for the coarsest possible sampling, with position sampling enabled and tangent direction sampling disabled.

Parameters

tolerance Maximum bound on the distance between the approximating polygon, formed by connecting the sample positions, and the true curve.

spacing Maximum bound on the arc length between successive sample points along a shape.

maxPoints Maximum number of sample points that may be taken along a single curved element of a primitive shape.

computeTangents Specifies whether or not to compute and return a tangent vector at each sample point (default = false).

duplicateCorners

Specifies how to handle points where either 1) two distinct contours meet (as in a contour tree), or 2) the tangent direction is discontinuous within a single contour (as at the corner of a rectangle). See *duplicateCorners* on page 2825 for more information.

Throws

ccShapesError::BadParams

tolerance is less than 0.0,
spacing is less than 0.0, or
maxPoints is less than 2.0.

Operators

operator==

```
bool operator==(const ccSampleParams &rhs) const;
```

Returns true if *rhs* is equal to this **ccSampleParams** object.

Parameters

rhs The other **ccSampleParams** object.

operator!=

```
bool operator!=(const ccSampleParams &rhs) const;
```

Returns true if *rhs* is not equal to this **ccSampleParams** object.

Parameters

rhs The other **ccSampleParams** object.

Public Member Functions

tolerance

```
double tolerance() const;
```

```
void tolerance(const double val);
```

- `double tolerance() const;`

Gets the tolerance bound. This is a maximum bound on the distance between the approximating polygon, formed by connecting the sample positions, and the true curve. If meeting the tolerance bound would cause more than **maxPoints()** to be generated along a curved element of a primitive shape, the **sample()** method throws *ccShapesError::SampleOverflow*. A tolerance value of 0.0 (the default) is special, indicating that there is no maximum tolerance bound; use this value to generate the coarsest possible sampling. The tolerance bound is only significant for contours that contain curves.

- `void tolerance(const double val);`

  Sets the tolerance bound.

  Parameters

  val The tolerance value. Must be greater than or equal to 0.0.

  Throws

  ccShapesError::BadParams

  val is less than 0.0.

spacing

```
double spacing() const;

void spacing(double val);
```

- `double spacing() const;`

  Gets the spacing bound. This is a maximum bound on the arc length between successive sample points along a shape. If meeting the spacing bound would cause more than **maxPoints()** to be generated along a curved element of a primitive shape, the **sample()** method throws *ccShapesError::SampleOverflow*. A spacing value of 0.0 (the default) is a special flag indicating that there is no maximum spacing bound; use this value to generate the coarsest possible sampling.

- `void spacing(double val);`

  Sets the spacing bound.

  Parameters

  val The spacing bound. Must be greater than or equal to 0.0.

  Throws

  ccShapesError::BadParams

  val is less than 0.0.

maxPoints

```
c_Int32 maxPoints() const;

void maxPoints(c_Int32 val);
```

- `c_Int32 maxPoints() const;`

  Gets the maximum points value. This value is the maximum number of sample points that may be taken along a single element of a primitive shape. If the spacing or tolerance bounds are set in a manner that requires more than **maxPoints()** samples, the **sample()** method will throw *ccShapesError::SampleOverflow*.

- `void maxPoints(c_Int32 val);`

  Sets the max points value.

  val The maximum points value. Must be greater than or equal to 2.

  Throws

  ccShapesError::BadParams

  val is less than 2.

computeTangents

```
bool computeTangents() const;

void computeTangents(bool flag);
```

- `bool computeTangents() const;`

  Gets the compute tangents flag. This flag determines whether or not tangent direction sampling should be done along with the standard position sampling.

  Notes

  If **computeTangents()** is true, the **sample()** method ignores shapes for which **hasTangent()** is false.

- `void computeTangents(bool flag);`

  Sets the compute tangents flag. This flag determines whether or not tangent direction sampling should be done along with the standard position sampling.

  Parameters

  flag If true, the **sample()** method will compute tangents in addition to sample points, ignoring shapes for which **hasTangent()** is false.

  Notes

  Do not change this value between invocations of **sample()** if results are collected in a single **ccSampleResult** object.

  See **ccShape::sample()** for more information.

duplicateCorners

```
bool duplicateCorners() const;

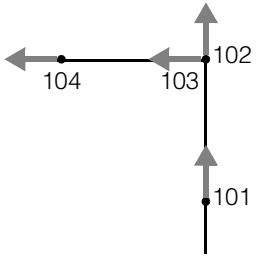
void duplicateCorners(bool flag);
```

- `bool duplicateCorners() const;`

  Gets the *duplicateCorners* flag. This flag determines how to handle points where two distinct contours meet (as in a **ccContourTree**) or the tangent direction is discontinuous within a single contour (as at the corner of a rectangle).

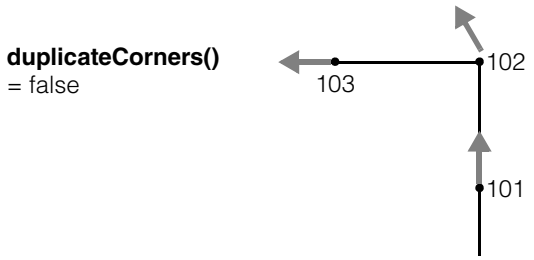
  - If **duplicateCorners()** is true, two samples are generated at each junction. The two samples share the same position, but the first sample takes its tangent direction from the first curve, and the second sample takes its tangent direction from the second curve. For example, in the following figure in which **duplicateCorners()** is true, there are two sample points (102 and 103) at the corner and each takes its tangent direction from one of the two line segments that join at the corner.

    duplicateCorners()
    = true

    If **duplicateCorners()** is true, point samples are duplicated at the corners even if **computeTangents()** is false.

- If **duplicateCorners()** is false, only one sample is generated at the junction, and the tangent direction is the average of the tangent directions from the two curves. For example, in the following figure in which **duplicateCorners()** is false, there is only one sample point (102) at the corner and its tangent direction is the average of the tangent directions of the two line segments that join at the corner.



- `void duplicateCorners(bool flag);`

  Sets the duplicate corners flag.

  flag The duplicate corners flag. Must be a boolean value.

uniformMode

```
bool uniformMode() const;
void uniformMode(bool flag);
```

- `bool uniformMode() const;`

  For Cognex internal use only.

- `void uniformMode(bool flag);`

  For Cognex internal use only.

ccSampleProp

```
#include <ch_cvl/prop.h>

class ccSampleProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class encapsulates subsampling properties applied to acquired images. Image x and y subsampling are performed independently. Subsampling can be used to reduce the image size and may also result in faster acquires. Subsample values specify the ratio of full frame pixels to sampled pixels. For example, an x subsample of 8 specifies an 8:1 reduction in the number of pixels along the x (horizontal) axis. The y subsample specifies a reduction of pixels along the y (vertical) axis.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccSampleProp

```
ccSampleProp();

ccSampleProp(
  c_Int32 subSampleX,
  c_Int32 subSampleY);

virtual ~ccSampleProp() {};
```

- `ccSampleProp();`

  Default constructor.

- ```
  ccSampleProp(
    c_Int32 subSampleX,
    c_Int32 subSampleY);
  ```

  Constructor with specified subsampling.

  Parameters

  subSampleX The x subsample value.

  subSampleY The y subsample value.

  Notes

  An invalid *subSampleX defaults to* 1, no x subsampling. Also, an invalid `subSampleY` defaults to 1, no y subsampling.

- ```
  virtual ~ccSampleProp() {};
  ```

  Destructor.

Operators

operator== `bool operator==(const ccSampleProp& that) const;`

Comparison operator. Returns true if *that* = this object. Returns false otherwise.

Parameters

that A **ccSampleProp** object to compare with this one.

operator!= `bool operator!=(const ccSampleProp& that) const;`

Comparison operator. Returns false if *that* = this object. Returns true otherwise.

Parameters

that A **ccSampleProp** object to compare with this one.

Public Member Functions

sampleX

```
void sampleX(c_Int32 v);
c_Int32 sampleX() const;
```

- `void sampleX(c_Int32 v);`

  Sets a new subsample x-value.

  Notes

  An invalid subsample x-value sets *v* = 1, no x subsampling.

  Parameters

v	The new subsample value.

- `c_Int32 sampleX() const;`

  Returns the current subsample x-value.

sampleY

```
void sampleY(c_Int32 v);
c_Int32 sampleY() const;
```

- `void sampleY(c_Int32 v);`

  Sets a new subsample y-value.

  Parameters

v	The new subsample value.

  Notes

  An invalid subsample y-value sets *v* = 1, no y subsampling.

- `c_Int32 sampleY() const;`

  Returns the current subsample y-value.

Constants

defaultSampleX `static const c_Int32 defaultSampleX;`

Default = 1, no subsampling.

defaultSampleY `static const c_Int32 defaultSampleY;`

Default = 1, no subsampling.

ccSampleResult

```
#include <ch_cvl/shapbase.h>

class ccShape::ccSampleResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

The **ccSampleResult** class describes the result of a **ccShape::sample()** operation. This is a nested class defined inside of **ccShape**.

When sampling a shape with multiple contours, such as an annulus or hierarchical shape, there are multiple *chains* of sampled positions and tangents. You can retrieve these positions and tangents as flat (one-dimensional) vectors from a **ccSampleResult** object using the **positions()** and **tangents()** methods, respectively.

The **breaks()** method returns a vector that describes how these composite vectors are partitioned into individual chains: each element of the *breaks* vector indexes the starting element of a distinct chain in the *positions* and *tangents* vectors. Thus, the size of the *breaks* vector equals the number of distinct chains. This scheme facilitates effective pre-allocation of storage for position and tangent samples.

The *closedFlags* vector always has the same number of elements as the *breaks* vector. The *i*th element of *closedFlags* indicates whether the chain indexed by the *i*th element of *breaks* is an open or closed chain.

Constructors/Destructors

ccSampleResult

```
ccSampleResult();
```

Default constructor. Constructs an empty **ccSampleResult** object.

Operators

operator==

```
bool operator==(const ccSampleResult &rhs) const;
```

Returns true if *rhs* is equal to this **ccSampleResult** object.

Parameters

rhs	The other **ccSampleResult** object.

operator!=

```
bool operator!=(const ccSampleResult &rhs) const;
```

Returns true if *rhs* is not equal to this **ccSampleResult** object.

Parameters

rhs The other **ccSampleResult** object.

Public Member Functions

reset

```
void reset();
```

Removes all position samples, tangent direction samples, and break indices stored within this **ccSampleResult**.

numChains

```
c_Int32 numChains() const;
```

Returns the number of distinct chains within this **ccSampleResult**.

positions

```
const cmStd vector<cc2Vect> &positions() const;
cmStd vector<cc2Vect> &positions();
```

- ```
  const cmStd vector<cc2Vect> &positions() const;
  ```

  Returns the positions computed from one or more sample operations.

  Notes

  If this **ccSampleResult** object has not been used in any sampling operations, the returned vector will be empty. If this **ccSampleResult** has been used in multiple sampling operations, the returned vector will contain the concatenation of sampled positions from all of the operations.

- ```
  cmStd vector<cc2Vect> &positions();
  ```

  This overload is for Cognex internal use only.

tangents

```
const cmStd vector<ccRadian> &tangents() const;
cmStd vector<ccRadian> &tangents();
```

- ```
  const cmStd vector<ccRadian> &tangents() const;
  ```

  Returns the tangent directions computed from one or more sample operations.
  ```

Notes

If this **ccSampleResult** object has not been used in any sampling operations, the returned vector will be empty. If this **ccSampleResult** has been used in multiple sampling operations, this vector will contain the concatenation of sampled tangent directions from all of the operations.

- `cmStd vector<ccRadian> &tangents();`

  This overload is for Cognex internal use only.

breaks

```
const cmStd vector<c_Int32> &breaks() const;
cmStd vector<c_Int32> &breaks();
```

- `const cmStd vector<c_Int32> &breaks() const;`

  Returns the break indices computed from one or more sample operations. The break indices impose a structure on the flat vectors returned by **positions()** and **tangents()**. See comments at the beginning of this class for details.

  Notes

  If this **ccSampleResult** object has not been used in any sampling operations, the returned vector will be empty. If this **ccSampleResult** has been used in multiple sampling operations, this vector will contain the concatenation of breakpoints from all of the operations.

- `cmStd vector<c_Int32> &breaks();`

  This overload is for Cognex internal use only.

closedFlags

```
const cmStd vector<bool> &closedFlags() const;
cmStd vector<bool> &closedFlags();
```

- `const cmStd vector<bool> &closedFlags() const;`

  Returns the closed flags computed from one or more sample operations. There is one closed flag for each sample chain indicating whether or not that chain was obtained from a closed contour. See comments at the top of this class for details on chains.

Notes

If this **ccSampleResult** object has not been used in any sampling operations, this vector will be empty. If this **ccSampleResult** has been used in multiple sampling operations, this vector will contain the concatenation of closed flags from all of the operations.

- `cmStd vector<bool> &closedFlags();`

  This overload is for Cognex internal use only.

getPolylines

```
void getPolylines(cmStd vector<ccPolyline> &polylines)
  const;
```

Computes a vector of polylines corresponding to the position samples of this **ccSampleResult**. Each polyline corresponds to one chain of samples.

Parameters

polylines The vector of polylines returned.

positionsReserve

```
void positionsReserve(c_Int32 n);
```

Pre-allocates enough memory to store *n* position samples within this **ccSampleResult** object.

Parameters

n The number of position samples to allocate memory for.

Notes

This class dynamically re-allocates memory as needed. It is never necessary for you to use this method. It is provided strictly as an optimization.

If *n* is less than the current size of the *positions* vector, this method has no effect.

positionsTangentsReserve

```
void positionsTangentsReserve(c_Int32 n);
```

Pre-allocates enough memory to store *n* position samples and *n* tangent direction samples within this **ccSampleResult** object.

Parameters

n The number of position samples and tangent direction samples to allocate memory for.

Notes

This class dynamically re-allocates memory as needed. It is never necessary for you to use this method. It is provided strictly as an optimization.

If *n* is less than the current size of the *positions* vector, this method has no effect.

chainsReserve `void chainsReserve(c_Int32 n);`

Pre-allocates enough memory to store *n* break indices and *n* closed flags within this **ccSampleResult** object.

Parameters

n	The number of break indices and closed flags to allocate memory for.

Notes

This class dynamically re-allocates memory as needed. It is never necessary for you to use this method. It is provided strictly as an optimization.

If *n* is less than the current size of the *positions* vector, this method has no effect.

ccSampleResult

ccSceneAngleFinderIIResult

```
#include <ch_cvl/scnangle.h>

class ccSceneAngleFinderIIResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single Scene Angle Finder-II tool result.

Note You should not instantiate this class directly.

Constructors/Destructors

ccSceneAngleFinderIIResult

```
ccSceneAngleFinderIIResult(double signalToNoise = 0,
  double rawScore = 0, const ccDegree& angle = ccDegree(0));
```

Constructs a **ccSceneAngleFinderIIResult**.

Parameters

signalToNoise	The signal-to-noise value (in dB) of this result.
rawScore	The raw score of this result
angle	The angle of the predominant feature boundary in client coordinates.

Operators

operator==

```
bool operator==(const ccSceneAngleFinderIIResult& rhs)
  const;
```

Two **ccSceneAngleFinderIIResult**s are equal if all their members are equal.

Parameters

rhs	The **ccSceneAngleFinderIIResult** to compare to this one.

Public Member Functions

angle

```
ccDegree angle() const;
```

Returns the detected scene angle in the client coordinate system of the runtime image.

Notes

In unidirectional mode, the angle returned ranges from -90 to 90 degrees. In bidirectional mode, the angle returned ranges from -45 to 45 degrees.

rawScore

```
double rawScore() const;
```

Returns the raw score of this result.

signalToNoise

```
double signalToNoise() const;
```

Returns the signal to noise ratio of this result in dB.

ccSceneAngleFinderIIResultSet

```
#include <ch_cvl/scnangle.h>

class ccSceneAngleFinderIIResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a set of **ccSceneAngleFinderResultII** objects.

Constructors/Destructors

ccSceneAngleFinderIIResultSet

```
ccSceneAngleFinderIIResultSet();
```

Creates a **ccSceneAngleFinderIIResultSet** with no results and with its execution time set to 0.0.

Operators

operator==

```
bool operator== (const ccSceneAngleFinderIIResultSet& rhs)
  const;
```

Two **ccSceneAngleFinderIIResultSet**s are equal if all their members are equal.

Parameters

rhs The **ccSceneAngleFinderIIResultSet** to compare to this one.

Public Member Functions

time

```
double time() const;
```

Returns the time required to produce this result set in seconds.

numFound

```
c_Int32 numFound() const;
```

Returns the number of results in this **ccSceneAngleFinderIIResultSet**.

results

```
const cmStd vector<ccSceneAngleFinderIIResult>&
  results() const;
```

Returns a vector containing all of the individual results. The number of elements in the returned vector is returned by **numFound()**.

Notes

The results are sorted by raw score.

ccSceneAngleFinderIIRunParams

```
#include <ch_cvl/scnangle.h>

class ccSceneAngleFinderIIRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters for the Scene Angle Finder-II tool.

Enumerations

Interpolation `enum Interpolation`

This enumeration defines supported interpolation accuracy levels for the Scene Angle Finder-II tool.

Value	Meaning
eNormal	Performs image interpolation using standard bilinear interpolation.
eHighPrecision	Performs image interpolation using an advanced interpolation method.
kDefaultInterpolation	The default interpolation method, defined in *scnangle.h*.

Constructors/Destructors

ccSceneAngleFinderIIRunParams

```
ccSceneAngleFinderIIRunParams(c_Int32 maxNumResults = 1,
  const ccDegree& startAngle = ccDegree(-90),
  const ccDegree& endAngle = ccDegree(90),
  c_Int32 initialSampling = 4, c_Int32 finalSampling = 1,
  double acceptThreshold = 0.5,
  double lowResThreshold = 50.0,
```

```
bool bidirectional = false,
ccSceneAngleFinderIIRunParams::Interpolation precision =
ccSceneAngleFinderIIRunParams::kDefaultInterpolation);
```

Constructs a **ccSceneAngleFinderIIRunParams** using the supplied parameters.

Parameters

maxNumResults The maximum number of results to compute.

startAngle The start of the angle range to consider.

endAngle The end of the angle range to consider

initialSampling The subsampling value for the preliminary evaluation.

finalSampling The subsampling value for the final measurement.

acceptThreshold
The minimum score for a valid result, expressed as the signal to noise ratio in dB.

lowResThreshold
The minimum raw score threshold. Only preliminary results with raw scores above this value are evaluated further. Raw scores can be in the range 0.0 through the square of the maximum pixel value. In most cases, the default value of 50 is appropriate.

bidirectional If true, all features are assumed to lie at 90° angles to each other and the returned angle is in the range -45° through 45°. If false, then features are assumed to lie at 180° angles to each other and the returned angle is in the range -90° through 90°.

precision The interpolation precision to use. Using the *eHighPrecision* interpolation method produces more accurate results but takes slightly longer. The default is *eNormal*.

precision must be one of the following values:

ccSceneAngleFinderIIRunParams::eNormal
ccSceneAngleFinderIIRunParams::eHighPrecision
ccSceneAngleFinderIIRunParams::kDefault

Operators

operator==

```
bool operator== (const ccSceneAngleFinderIIRunParams& rhs)
 const;
```

Two **ccSceneAngleFinderIIRunParams**s are equal if all their members are equal.

Parameters

rhs The **ccSceneAngleFinderIIRunParams** to compare to this one.

Public Member Functions

maxNumResults

```
c_Int32 maxNumResults() const;
void maxNumResults(c_Int32 maxNum);
```

- `c_Int32 maxNumResults() const;`

  Returns the maximum number of results the tool will compute.

- `void maxNumResults(c_Int32 maxNum);`

  Sets the maximum number of results specified for this **ccSceneAngleFinderIIRunParams**. No more than the specified number of results are computed when the tool is run.

  Parameters

  maxNum The maximum number of results.

  Throws

  ccSceneAngleFinderDefs::BadParams
  maxNum is less than or equal to 0.

bidirectional

```
bool bidirectional() const;
```

Returns true if this **ccSceneAngleFinderIIRunParams** specifies bidirectional mode, false if it specifies unidirectional mode.

In bidirectional mode, angles at θ and θ+ 90° are evaluated together. The maximum angle span is 90°. In unidirectional mode, the maximum angle span is 180°.

startAngle

```
ccDegree startAngle() const;
```

Returns the start of the angle range specified for this **ccSceneAngleFinderIIRunParams**.

endAngle

```
ccDegree endAngle() const;
```

Returns the end of the angle range specified for this **ccSceneAngleFinderIIRunParams**.

ccSceneAngleFinderIIRunParams

angleSpan

```
void angleSpan(const ccDegree& degreeStartAngle,
  const ccDegree& degreeEndAngle, bool bidirectional);
```

Sets the angle span over which to search for the dominant scene angle or angles. The search will be conducted from the start angle to the end angle. Either or both angles can be negative.

Parameters

degreeStartAngle
The start angle in client coordinates.

degreeEndAngle
The end angle in client coordinates

bidirectional If true, this **ccSceneAngleFinderIIRunParams** is set to bidirectional mode. If false, it is set to unidirectional mode.

Notes

In unidirectional mode, all angles are mapped to the range -90° through 90°. In bidirectional mode, all angles are mapped to the range -45 through 45°.

The angles are specified in client coordinates.

Throws

ccSceneAngleFinderDefs::BadParams
startAngle is greater than or equal to *endAngle*; the specified angle range is greater than 90° in bidirectional mode; or the angle range is greater than 180° in unidirectional mode.

initialSampling

```
c_Int32 initialSampling() const;
```

Returns the subsampling rate for the initial phase of the analysis.

finalSampling

```
c_Int32 finalSampling() const;
```

Returns the subsampling rate for the final phase of the analysis.

setSampling

```
void setSampling(c_Int32 initialSampling,
  c_Int32 finalSampling);
```

Sets the subsampling rate for both phases of the analysis.

Parameters

initialSampling The subsampling rate for the initial phase of the analysis.

finalSampling The subsampling rate for the final phase of the analysis.

Throws

ccSceneAngleFinderDefs::BadParams

initialSampling or *finalSampling* is less than or equal to 0 or *initialSampling* is less than *finalSampling*

Notes

If *initialSampling* is equal to *finalSampling*, only the preliminary phase of analysis is performed.

---

acceptThreshold

```
double acceptThreshold() const;
void acceptThreshold(double accept);
```

- `double acceptThreshold() const;`

  Returns the accept threshold for this **ccSceneAngleFinderIIRunParams**.

- `void acceptThreshold(double accept);`

  Sets the accept threshold for this **ccSceneAngleFinderIIRunParams**. Only results with a signal to noise ratio that is greater than the accept threshold are returned by the tool.

  Parameters

  accept The accept threshold.

  Notes

  If you specify an accept threshold less than or equal to 0.0, then the tool returns all the results it finds, up to the maximum number you specify.

lowResThreshold

```
double lowResThreshold() const;
void lowResThreshold(double threshold);
```

- `double lowResThreshold() const;`

  Returns the low-resolution accept threshold.

- `void lowResThreshold(double threshold);`

  Sets the low-resolution accept threshold. Only results with raw scores greater than this threshold will be evaluated during the final phase of analysis.

Parameters

threshold The threshold.

Notes

If the low resolution threshold is less than or equal to 0, it will use any relative maximum raw score as starting points for the search in the final pass.

---

accuracyMode

```
ccSceneAngleFinderIIRunParams::Interpolation
  accuracyMode() const;

void accuracyMode(
  ccSceneAngleFinderIIRunParams::Interpolation precision);
```

- ```
  ccSceneAngleFinderIIRunParams::Interpolation
    accuracyMode() const;
  ```

  Returns the interpolation mode used by this **ccSceneAngleFinderIIRunParams**. The returned value is one of the following:

  ccSceneAngleFinderIIRunParams::eNormal
  ccSceneAngleFinderIIRunParams::eHighPrecision

- ```
  void accuracyMode(
    ccSceneAngleFinderIIRunParams::Interpolation precision);
  ```

  Sets the interpolation mode used by this **ccSceneAngleFinderIIRunParams**.

  Parameters

  precision The interpolation precision to use. Using the *eHighPrecision* interpolation method produces more accurate results but takes slightly longer. The default is *eNormal*.

  precision must be one of the following values:

  ccSceneAngleFinderIIRunParams::eNormal
  ccSceneAngleFinderIIRunParams::eHighPrecision
  ccSceneAngleFinderIIRunParams::kDefault

ccSceneAngleFinderResult

```
#include <ch_cvl/scnangle.h>

class ccSceneAngleFinderResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class contains a single Scene Angle Finder tool result.

Note You should not instantiate this class directly.

Constructors/Destructors

ccSceneAngleFinderResult

```
ccSceneAngleFinderResult(
  double dfilteredPeakMagnitude = 0,
  ccDegree degreeAngle = ccDegree(0));
```

Constructs a **ccSceneAngleFinderResult**.

Parameters

dfilteredPeakMagnitude
The peak magnitude.

degreeAngle The scene angle.

Public Member Functions

angle

```
ccDegree angle() const;
```

Returns the scene angle in the client coordinate system of the run-time image.

filteredPeakMagnitude

```
double filteredPeakMagnitude() const;
```

Returns the magnitude of the peak in the filtered histogram.

ccSceneAngleFinderResult

ccSceneAngleFinderResultSet

```
#include <ch_cvl/scnangle.h>

class ccSceneAngleFinderResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a set of **ccSceneAngleFinderResult** objects.

Constructors/Destructors

ccSceneAngleFinderResultSet

```
ccSceneAngleFinderResultSet();
```

Creates a **ccSceneAngleFinderResultSet** with no results.

Public Member Functions

time

```
double time() const;
```

Returns the amount time required to produce this result set in seconds.

numFound

```
c_Int32 numFound() const;
```

Returns the number of **ccSceneAngleFinderResult**s in this **ccSceneAngleFinderResultSet**.

histogram

```
const cmStd vector<c_UInt32>& histogram() const;
```

Returns the unfiltered edge angle histogram.

results

```
const cmStd vector<ccSceneAngleFinderResult>& results()
  const;
```

Return all of the results from the Scene Angle Finder tool.

ccSceneAngleFinderResultSet

ccSceneAngleFinderRunParams

```
#include <ch_cvl/scnangle.h>

class ccSceneAngleFinderRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters for the Scene Angle Finder tool.

Constructors/Destructors

ccSceneAngleFinderRunParams

```
ccSceneAngleFinderRunParams(c_Int32 nToFind = 1,
  ccDegree degreeStartAngle = ccDegree(-90),
  ccDegree degreeEndAngle = ccDegree(90),
  c_Int32 numberOfFolds = 2, c_Int32 binFilterWidth = 5,
  c_Int32 nContrastThreshold = -1, c_Int32 nSubsampling = 2,
  c_Int32 nSmoothingOffset = -1);
```

Constructs a **ccSceneAngleFinderRunParams** with the specified parameters.

For more information on the individual parameters, see the individual member functions in this class.

Parameters

nToFind The number of scene angle peaks to find.

degreeStartAngle The start of the angle span in which to search for peaks.

degreeEndAngle The end of the angle span in which to search for peaks.

numberOfFolds The number of folds to apply to the edge angle histogram.

binFilterWidth The width of the kernel used to filter the edge angle histogram.

nContrastThreshold The minimum edge magnitude required for including an edge in the edge angle histogram.

nSubsampling The subsampling value applied to the image before computing the edge angle image.

nSmoothingOffset
The smoothing value to apply to the image before computing the edge angle is equal to *nSubsampling*+*nSmoothingOffset*.

Public Member Functions

maxNumResults

```
c_Int32 maxNumResults() const;
void maxNumResults(c_Int32 nToFind);
```

- `c_Int32 maxNumResults() const;`

  Returns the maximum number of scene angle peaks this **ccSceneAngleFinderRunParams** is configured to find.

- `void maxNumResults(c_Int32 nToFind);`

  Sets the maximum number of scene angle peaks this **ccSceneAngleFinderRunParams** is configured to find.

  Parameters
  nToFind The number of peaks to find.

  Throws
  ccSceneAngleFinderDefs::BadParams
  nToFind is less than 1.

startAngle `ccDegree startAngle() const;`

Returns the start angle of the angle span.

endAngle `ccDegree endAngle() const;`

Returns the end angle of the angle span.

angleSpan

```
void angleSpan(const ccDegree& degreeStartAngle,
  const ccDegree& degreeEndAngle);
```

Sets the angle span over which to search for the dominant scene angles. The search will be conducted from the start angle to the end angle. Either or both angle can be negative, however a positive angle increment will be used with wrapping from 360 to zero.

The size of the angle span must be less than or equal to the number of degrees per fold. You can call **angleSpanAndNumberOfFolds()** to change the degrees per fold at the same time as the angle span.

Parameters

degreeStartAngle
The start angle of the angle span, expressed as the angle from the client coordinate x-axis to the positive gradient direction of the feature.

degreeEndAngle The end angle of the angle span, expressed as the angle from the client coordinate x-axis to the positive gradient direction of the feature.

Notes

The default start angle is -90; end angle is 90

If either *degreeStartAngle* or *degreeEndAngle* is greater than 180°, then the returned scene angle is mapped to the range 0° to 360°. If both *degreeStartAngle* and *degreeEndAngle* are less than or equal to 180°, then the returned scene angle is mapped to the range -180° to +180°.

Throws

ccSceneAngleFinderDefs::BadParams
The angle span is greater than the number of degrees per fold, or the difference between *degreeStartAngle* and *degreeEndAngle* is greater than 360°.

numberOfFolds

```
c_Int32 numberOfFolds() const;
```

Returns the number of folds configured for this **ccSceneAngleFinderRunParams**.

angleSpanAndNumberOfFolds

```
void angleSpanAndNumberOfFolds(
  const ccDegree& degreeStartAngle,
  const ccDegree& degreeEndAngle,
  c_Int32 numberOfFolds);
```

Sets the number of folds and the angle span. The number of folds controls how many times the raw edge angle histogram is folded upon itself to produce the edge angle histogram used to score scene angles.

You should set the fold value to match the number of reinforcing edges in a scene. For example, if the image contains rectilinear features then you should set the fold value to 4.

The angle span must be less than or equal to the number of degrees per fold. The number of degrees per fold is computed by dividing 360° by the fold value.

Parameters

degreeStartAngle
The start of the angle span in which to search for peaks.

degreeEndAngle The end of the angle span in which to search for peaks.

numberOfFolds The number of folds to apply to the edge angle histogram.

Notes

If the polarity of the edges in the image can vary, set the fold value to be greater than or equal to 2. This ensures that edge polarity is not considered.

Throws

ccSceneAngleFinderDefs::BadParams
The angle span is greater than the number of degrees per fold, or *numberOfFolds* is not 2, 4, 8, 16, or 32.

filterWidth

```
c_Int32 filterWidth() const;

void filterWidth(c_Int32 binFilterWidth);
```

- `c_Int32 filterWidth() const;`

  Returns the width of the one-dimensional filter applied to the folded edge angle histogram.

- `void filterWidth(c_Int32 binFilterWidth);`

  Sets the width of the one-dimensional filter applied to the folded edge angle histogram. The specified width must be odd. If an even width is supplied, it is incremented by 1.

  Increasing the filter width increases the smoothing of the peaks in the edge angle histogram. This can improve accuracy when many closely grouped angle peaks are present. Increasing the filter width too much can blur distinct peaks into one peak. The default filter width is 5.

  Parameters

  binFilterWidth The filter width, in histogram pixels.

  Throws

  ccSceneAngleFinderDefs::BadParams
  binFilterWidth is less than 1.

contrastThreshold

```
c_Int32 contrastThreshold() const;

void contrastThreshold(c_Int32 nContrastThreshold);
```

- `c_Int32 contrastThreshold() const;`

  Returns the minimum contrast required before an edge is added to the edge angle histogram.

- `void contrastThreshold(c_Int32 nContrastThreshold);`

  Sets the minimum contrast required before an edge is added to the edge angle histogram. Set the threshold to be 1 or 2 more than number of grey levels of contrast in the direction of the gradient in the edges of interest.

  For horizontal and vertical edges the magnitude will be the grey level difference. For edges which are not horizontal or vertical the magnitude is slightly greater than the number of grey level difference.

  Specify -1 for the contrast threshold to use the Edge tool's default threshold.

  Parameters

  nContrastThreshold
  The contrast threshold.

  Throws

  ccSceneAngleFinderDefs::BadParams
  nContrastThreshold is less than -1.

subsampling

```
c_Int32 subsampling() const;
```

Returns the subsampling value applied to the input image before the edge angle image is computed.

smoothingOffset

```
c_Int32 smoothingOffset() const;
```

Returns the smoothing offset applied to the input image before the edge angle image is computed. The input image is smoothed using a smoothing value equal to the subsampling value plus the smoothing offset before the edge angle image is computed.

subsamplingAndSmoothing

```
void subsamplingAndSmoothing(c_Int32 nSubsampling,
  c_Int32 nSmoothingOffset = -1);
```

Sets the subsampling and smoothing values applied to the input image before the edge angle image is computed. The image is subsampled by the subsampling value, then smoothed using a smoothing value equal to the subsampling value plus the smoothing offset before the edge angle image is computed.

Parameters

nSubsampling The subsampling value applied to the image before computing the edge angle image.

nSmoothingOffset
The smoothing value to apply to the image before computing the edge angle is equal to *nSubsampling*+*nSmoothingOffset*.

Notes

The default subsampling value is 2; the default smoothing offset is -1.

Throws

ccSceneAngleFinderDefs::BadParams
nSubsampling is less than 1 or greater than 25, or *nSubsampling*+*nSmoothingOffset* is less than 0 or greater than 24.

ccScoreContrast

```
#include <ch_cvl/clpscore.h>

class ccScoreContrast: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the contrast of an edge candidate to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the difference in normalized pixel values across the edge.

To define the break points for the function, specify values in the constructor. To modify the set points, use the member functions in **ccScoreOneSided**.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Constructors/Destructors

ccScoreContrast

```
ccScoreContrast (double x0 = 255.0, double x1 = 0.0,
  double xc = 0.0, double y0 = 1.0, double y1 = 0.0);
```

Constructs a **ccScoreContrast** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScoreContrast** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScoreContrast**.
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScoreContrast** and for the supplied **ccScoreContrast**.

Parameters

that The **ccScoreContrast** to compare to this one

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccScoreOneSided

```
#include <ch_cvl/clpscore.h>

class ccScoreOneSided : public ccCaliperScore;
```

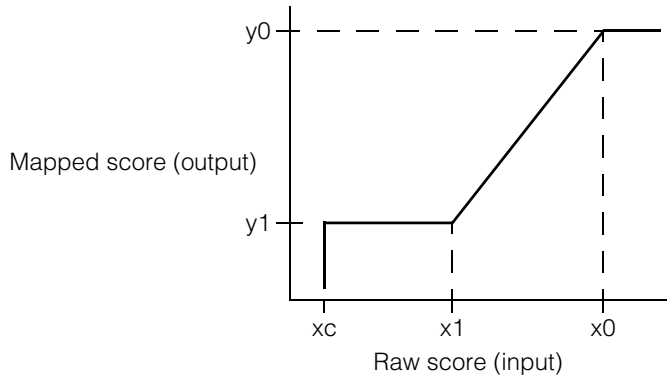
Class Properties

Copyable	Yes
Derivable	By Cognex-supplied classes only
Archiveable	Complex

A class from which concrete classes such as **ccScoreContrast** and **ccScorePosition** are derived.

You use the public member functions in this class to change the points that define a scoring function used by one of the concrete classes.

A one-sided scoring function maps a raw input score to a mapped output score using a function defined by the points shown in the following figure:



Constructors/Destructors

ccScoreOneSided

```
ccScoreOneSided (double x0 = 0.0, double x1 = 1.0,
  double xc = 1.0, double y0 = 1.0, double y1 = 0.0);
```

Constructs a **ccScoreOneSided** using the supplied values to construct the scoring function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
 const;
```

Tests this **ccScoreOneSided** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScoreOneSided**
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScoreOneSided** and for the supplied **ccScoreOneSided**

Parameters

that	The **ccScoreOneSided** to compare to this one.

Public Member Functions

x0

```
double x0 () const;

void x0 (double x0);
```

- `double x0 () const;`

  Returns the *x0* value.

- `void x0 (double x0);`

  Sets the *x0* value.

  Parameters

x0	The value to set.

x1

```
double x1 () const;
void x1 (double x1);
```

- `double x1 () const;`

  Returns the *x1* value.

- `void x1 (double x1);`

  Sets the *x1* value.

  Parameters

x1	The value to set *x1* must be between *x0* and *xc*.

xc

```
double xc () const;
void xc (double xc);
```

- `double xc () const;`

  Returns the *xc* value.

- `void xc (double xc);`

  Sets the *xc* value.

  Parameters

xc	The value to set.

y0

```
double y0 () const;
void y0 (double y0);
```

- `double y0 () const;`

  Returns the *y0* value.

- `void y0 (double y0);`

  Sets the *y0* value.

Parameters

y0 The value to set. *y0* must be greater than or equal to *y1* and must be between 0.0 and 1.0.

Throws

ccCaliperDefs::BadParams
y0 is outside the range 0.0 to 1.0.

---

y1

```
double y1 () const;
void y1 (double y1);
```

- `double y1 () const;`

  Returns the *y1* value.

- `void y1 (double y1);`

  Sets the *y1* value.

  Parameters

  y1 The value to set. *y1* must be less than or equal to *y0* and must be between 0.0 and 1.0.

  Throws

  ccCaliperDefs::BadParams
  y1 is outside the range 0.0 to 1.0.

ccScorePosition

```
#include <ch_cvl/clpscore.h>

class ccScorePosition: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the position of an edge candidate to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the absolute value of the position at which this edge was found.

To define the break points for the function, specify values in the constructor. To modify the set points, use the member functions in **ccScoreOneSided**.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Constructors/Destructors

ccScorePosition

```
ccScorePosition (double x0 = 0.0, double x1 = 100.0,
  double xc = 10000.0, double y0 = 1.0, double y1 = 0.1);
```

Constructs a **ccScorePosition** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Notes

To *change* the value of any point in the scoring function, use the public member functions of the **ccScoreOneSided** class from which this class is derived.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScorePosition** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScorePosition**.
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScorePosition** and for the supplied **ccScorePosition**.

Parameters

that	The **ccScorePosition** to compare to this one

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges	The number of edges.

ccScorePositionNeg

```
#include <ch_cvl/clpscore.h>

class ccScorePositionNeg: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the position of an edge candidate to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the position at which this edge was found.

To define the break points for the function, specify values in the constructor. To modify the set points, use the member functions in **ccScoreOneSided**.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Constructors/Destructors

ccScorePositionNeg

```
ccScorePositionNeg (double x0 = -100.0, double x1 = 100.0,
  double xc = 10000.0, double y0 = 1.0, double y1 = 0.0);
```

Constructs a **ccScorePositionNeg** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Notes

To *change* the value of any point in the scoring function, use the public member functions of the **ccScoreOneSided** class from which this class is derived.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScorePositionNeg** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScorePositionNeg**.
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScorePositionNeg** and for the supplied **ccScorePositionNeg**.

Parameters

that The **ccScorePositionNeg** to compare to this one

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccScorePositionNorm

```
#include <ch_cvl/clpscore.h>

class ccScorePositionNorm: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the normalized position of an edge candidate to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the absolute value of the position at which this edge pair was found, divided by the nominal size of the edge pair.

Unlike **ccScorePosition**, this class normalizes the edge position by dividing it by the width of the edge model.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Note This class can only be used to score models with two edges.

Constructors/Destructors

ccScorePositionNorm

```
ccScorePositionNorm (double x0 = 0.0, double x1 = 5.0,
  double xc = 5.0, double y0 = 1.0, double y1 = 0.0);
```

Constructs a **ccScorePositionNorm** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.

y1 The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Notes

To *change* the value of any point in the scoring function, use the public member functions of the **ccScoreOneSided** class from which this class is derived.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScorePositionNorm** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScorePositionNorm**.
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScorePositionNorm** and for the supplied **ccScorePositionNorm**.

Parameters

that The **ccScorePositionNorm** to compare to this one

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccScorePositionNormNeg

```
#include <ch_cvl/clpscore.h>

class ccScorePositionNormNeg: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the normalized position of an edge candidate to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the position at which this edge pair was found, divided by the nominal size of the edge pair.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Note This class can only be used to score models with two edges.

Constructors/Destructors

ccScorePositionNormNeg

```
ccScorePositionNormNeg (double x0 = -5.0, double x1 = 5.0,
  double xc = 5.0, double y0 = 1.0, double y1 = 0.0);
```

Constructs a **ccScorePositionNormNeg** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Notes

To *change* the value of any point in the scoring function, use the public member functions of the **ccScoreOneSided** class from which this class is derived.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScorePositionNormNeg** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScorePositionNormNeg**.
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScorePositionNormNeg** and for the supplied **ccScorePositionNormNeg**.

Parameters

that The **ccScorePositionNormNeg** to compare to this one

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccScoreSizeDiffNorm

```
#include <ch_cvl/clpscore.h>

class ccScoreSizeDiffNorm: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the normalized difference between the size of an edge candidate and the size of the edge model to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the absolute difference between the nominal size of the edge pair and the found size of the edge pair, divided by the nominal size of the edge pair.

Unlike **ccScoreSizeDiff**, the difference between the expected size and the actual size is normalized by dividing it by the expected size.

To define the break points for the function, specify values in the constructor. To modify the set points, use the member functions in **ccScoreOneSided**.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Note This class can only be used to score edge models with two edges.

Constructors/Destructors

ccScoreSizeDiffNorm

```
ccScoreSizeDiffNorm (double x0 = 0.0, double x1 = 1.0,
  double xc = 1.0, double y0 = 1.0, double y1 = 0.0);
```

Constructs a **ccScoreSizeDiffNorm** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.

xc	The value for the *xc* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Notes

To *change* the value of any point in the scoring function, use the public member functions of the **ccScoreOneSided** class from which this class is derived.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScoreSizeDiffNorm** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScoreSizeDiffNorm**
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScoreSizeDiffNorm** and for the supplied **ccScoreSizeDiffNorm**

Parameters

that	The **ccScoreSizeDiffNorm** to compare to this one.

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges	The number of edges.

ccScoreSizeDiffNorm

ccScoreSizeDiffNormAsym

```
#include <ch_cvl/clpscore.h>

class ccScoreSizeDiffNormAsym: public ccScoreTwoSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreTwoSided** that lets you define a single scoring method that maps the normalized difference between the size of an edge candidate and the size of the edge model to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the difference between the nominal size of the edge pair and the found size of the edge pair, divided by the nominal size of the edge pair.

Like **ccScoreSizeDiffNorm**, the difference between the expected size and the actual size is normalized by dividing it by the expected size. The raw score produced by this class can be negative.

To define the break points for the function, specify values in the constructor. To modify the set points, use the member functions in **ccScoreTwoSided**.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Note This class can only be used to score edge models with two edges.

Constructors/Destructors

ccScoreSizeDiffNormAsym

```
ccScoreSizeDiffNormAsym (double x0 = 0.0, double x1 =-1.0,
  double xc =-1.0, double x0h = 0.0, double x1h = 2.0,
  double xch = 2.0, double y0 = 1.0, double y1 = 0.0,
  double y0h = 1.0, double y1h = 0.0);
```

Constructs a **ccScoreSizeDiffNormAsym** using the supplied points to define the mapping function.

Parameters

x0 The value for the *x0* point.

x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point
x0h	The value for the *x0h* point
x1h	The value for the *x1h* point. *x1h* must be between *x0h* and *xch*.
xch	The value for the *xch* point
y0	The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.
y1	The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.
y0h	The value for the *y0h* point. *y0h* must be greater than or equal to *y1h* and between 0.0 and 1.0.
y1h	The value for the *y1h* point. *y1h* must be less than or equal to *y0h* and between 0.0 and 1.0.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScoreSizeDiffNormAsym** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScoreSizeDiffNormAsym**.
- The values of *x0*, *x1*, *xc*, *x0h*, *x1h*, *xch*, *y0*, *y1*, *y0h*, and *y1h* are the same for this **ccScoreSizeDiffNormAsym** and for the supplied **ccScoreSizeDiffNormAsym**.

Parameters

that	The **ccScoreSizeDiffNormAsym** to compare to this one

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccScoreSizeDiffNormAsym

ccScoreSizeNorm

```
#include <ch_cvl/clpscore.h>

class ccScoreSizeNorm: public ccScoreOneSided;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived from **ccScoreOneSided** that lets you define a single scoring method that maps the normalized size of an edge candidate to a mapped score according to a scoring function that you define.

The raw input score for this scoring method is the found size of the edge pair divided by the nominal size of the edge pair.

Like **ccScoreSizeDiffNorm**, the expected size is normalized by dividing it by the expected size.

To define the break points for the function, specify values in the constructor. To modify the set points, use the member functions in **ccScoreOneSided**.

To use this class, create an instance of it, then pass a pointer to it in a vector of pointers to concrete classes to **ccCaliperRunParams::scoringMethods()**.

Note This class can only be used to score edge models with two edges.

Constructors/Destructors

ccScoreSizeNorm

```
ccScoreSizeNorm (double x0 = 5.0, double x1 = 0.0,
  double xc = 0.0, double y0 = 1.0, double y1 = 0.0) ();
```

Constructs a **ccScoreSizeNorm** using the supplied points to define the mapping function.

Parameters

x0	The value for the *x0* point.
x1	The value for the *x1* point. *x1* must be between *x0* and *xc*.
xc	The value for the *xc* point

y0 — The value for the *y0* point. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.

y1 — The value for the *y1* point. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

Notes

To *change* the value of any point in the scoring function, use the public member functions of the **ccScoreOneSided** class from which this class is derived.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScoreSizeNorm** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScoreSizeNorm**
- The values of *x0*, *x1*, *xc*, *y0*, and *y1* are the same for this **ccScoreSizeNorm** and for the supplied **ccScoreSizeNorm**

Parameters

that — The **ccScoreSizeNorm** to compare to this one.

Public Member Functions

score

For Cognex internal use only. Do not call this function.

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges	The number of edges.

ccScoreSizeNorm

ccScoreStraddle

```
#include <ch_cvl/clpscore.h>

class ccScoreStraddle: public ccCaliperScore;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A concrete class derived directly from **ccCaliperScore** that lets you score an edge pair candidate based on whether or not the two edges straddle the center point of the caliper window (or another, user-supplied origin). If the edges straddle the center point, the score is 1.0. If they do not, the score is 0.0.

To use this class, create an instance of it, then pass it in a vector of instances of concrete classes to **ccCaliperRunParams::scoringMethods()**.

Note This class can only be used to score edge models with two edges.

Constructors/Destructors

ccScoreStraddle

```
ccScoreStraddle();
```

Constructs a **ccScoreStraddle**.

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScoreStraddle** for equality with the supplied object. The function returns true if the supplied object is of type **ccScoreStraddle**.

Parameters

that The **ccScoreStraddle** to compare to this one

Public Member Functions

score For Cognex internal use only. Do not call this function.

ccScoreStraddle

clone

```
virtual ccCaliperScore* clone () const;
```

Allocates and returns a pointer to a duplicate of this object. This supports polymorphic copying.

willScore

```
virtual bool willScore (c_Int16 numEdges) const;
```

Return true if this scoring function can score candidates which contain the given number of edges.

Parameters

numEdges The number of edges.

ccScoreTwoSided

```
#include <ch_cvl/clpscore.h>

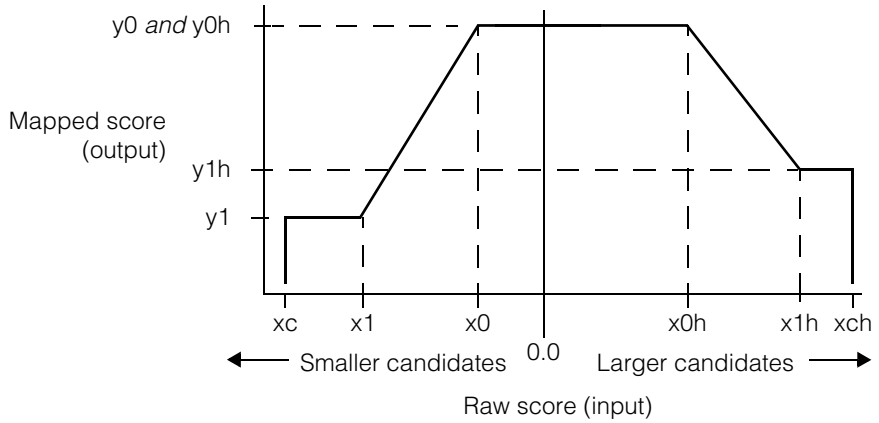
class ccScoreTwoSided: public ccCaliperScore;
```

Class Properties

Copyable	Yes
Derivable	By Cognex-supplied classes only
Archiveable	Complex

A base class that provides common implementations for all scoring methods using two-sided scoring functions.

A two-sided scoring function maps a raw input score to a mapped output score using a function defined by the points shown in the following figure:



Constructors/Destructors

ccScoreTwoSided

```
ccScoreTwoSided (double x0 = 0.0, double x1 = -1.0,
  double xc = -1.0, double x0h = 0.0, double x1h = 1.0,
  double xch = 1.0, double y0 = 1.0, double y1 = 0.0,
  double y0h = 1.0, double y1h = 0.0);
```

Constructs a **ccScoreTwoSided** using the supplied values to construct the scoring function.

Parameters

x0	The value for the *x0* point
x1	The value for the *x1* point
xc	The value for the *xc* point
x0h	The value for the *x0h* point
x1h	The value for the *x1h* point
xch	The value for the *xch* point
y0	The value for the *y0* point
y1	The value for the *y1* point
y0h	The value for the *y0h* point
y1h	The value for the *y1h* point

Operators

operator==

```
virtual bool operator== (const ccCaliperScore &that)
  const;
```

Tests this **ccScoreTwoSided** for equality with the supplied object. The function returns true if both of the following are true:

- The supplied object is of type **ccScoreTwoSided**.
- The values of *x0*, *x1*, *xc*, *x0h*, *x1h*, *xch*, *y0*, *y1*, *y0h*, and *y1h* are the same for this **ccScoreTwoSided** and for the supplied **ccScoreTwoSided**.

Parameters

that	The **ccScoreTwoSided** to compare to this one

Public Member Functions

x0

```
double x0 () const;
void x0 (double x0);
```

- `double x0 () const;`

  Returns the *x0* value.

- `void x0 (double x0);`

  Sets the *x0* value.

  Parameters

x0	The value to set. *x0* should be less than *x0h*, *x1h*, and *xch*.

x1

```
double x1 () const;
void x1 (double x1);
```

- `double x1 () const;`

  Returns the *x1* value.

- `void x1 (double x1);`

  Sets the *x1* value.

  Parameters

x1	The value to set. *x1* should be between *xc* and *x0* and less than *x0h*, *x1h*, and *xch*.

xc

```
double xc () const;
void xc (double xc);
```

- `double xc () const;`

  Returns the *xc* value.

- `void xc (double xc);`

  Sets the *xc* value.

Parameters

xc The value to set. *xc* should be less than *x0h*, *x1h*, and *xch*.

x0h

```
double x0h () const;
void x0h (double x0h);
```

- `double x0h () const;`

  Returns the *x0h* value.

- `void x0h (double x0h);`

  Sets the *x0h* value.

  Parameters

  x0h The value to set. *x0h* should be greater than *xc*, *x1*, and *x0*.

x1h

```
double x1h () const;
void x1h (double x1h);
```

- `double x1h () const;`

  Returns the *x1h* value.

- `void x1h (double x1h);`

  Sets the *x1h* value.

  Parameters

  x1h The value to set. *x1h* should be between *x0h* and *xch* and be greater than *xc*, *x1*, and *x0*.

xch

```
double xch () const;
void xch (double xch);
```

- `double xch () const;`

  Returns the *xch* value.

- `void xch (double xch);`

  Sets the *xch* value.

  Parameters

  xch The value to set. *xch* should be greater than *xc*, *x1*, and *x0*.

y0

```
double y0 () const;
void y0 (double y0);
```

- `double y0 () const;`

  Returns the *y0* value.

- `void y0 (double y0);`

  Sets the *y0* value.

  Parameters

  y0 The value to set. *y0* must be greater than or equal to *y1* and between 0.0 and 1.0.

  Throws

  ccCaliperDefs::BadParams
  y0 is outside the range 0.0 to 1.0.

y1

```
double y1 () const;
void y1 (double y1);
```

- `double y1 () const;`

  Returns the *y1* value.

- `void y1 (double y1);`

  Sets the *y1* value.

  Parameters

  y1 — The value to set. *y1* must be less than or equal to *y0* and between 0.0 and 1.0.

  Throws

  ccCaliperDefs::BadParams — *y1* is outside the range 0.0 to 1.0.

y0h

```
double y0h () const;
void y0h (double y0h);
```

- `double y0h () const;`

  Returns the *y0h* value.

- `void y0h (double y0h);`

  Sets the *y0h* value.

  Parameters

  y0h — The value to set. *y0h* must be greater than or equal to *y1h* and between 0.0 and 1.0.

  Throws

  ccCaliperDefs::BadParams — *y0h* is outside the range 0.0 to 1.0.

y1h

```
double y1h () const;
void y1h (double y1h);
```

- `double y1h () const;`

  Returns the *y1h* value.

- `void y1h (double y1h);`

  Parameters

  y1h — The value to set. *y1h* must be less than or equal to *y0h* and between 0.0 and 1.0.

  Throws

  ccCaliperDefs::BadParams — *y1h* is outside the range 0.0 to 1.0.

ccScoreTwoSided

ccSecurityInfo

```
#include <ch_cvl/secinfo.h>

class ccSecurityInfo;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

This class allows you to retrieve security information for a user-specified source within a Cognex vision system.

Constructors/Destructors

ccSecurityInfo

```
ccSecurityInfo(eInfoSource src = kSystem);

ccSecurityInfo(const ccBoard& board);

ccSecurityInfo(const ccSecurityInfo&);
```

- `ccSecurityInfo(eInfoSource src = kSystem);`

  Constructs a security information object that reports information on the selected source.

  Parameters

  src — The source for security information. The default value, *kSystem*, reports the complete set of licenses available to the application.

- `ccSecurityInfo(const ccBoard& board);`

  Constructs a security information object that reports information about the given board.

  Parameters

  board — The specific board for which licensing information is being reported.

- `ccSecurityInfo(const ccSecurityInfo& secInfo);`

  Copy constructor. Creates a security information object that is a copy of the given **ccSecurityInfo** object.

Parameters

secInfo The security information object to be copied.

~ccSecurityInfo

```
~ccSecurityInfo();
```

Destructor. Destroys this security information object.

Enumerations

eInfoSource

```
enum eInfoSource;
```

Values indicating the source for security information.

Value	Meaning
kSystem	Reports the set of licenses comprising the union of all licenses available on all detected Cognex hardware.
kReserved1	For Cognex internal use only.

Operators

operator=

```
ccSecurityInfo& operator=(const ccSecurityInfo& secInfo);
```

Assignment operator. Assigns the given security information object to this one.

Parameters

secInfo The security information object to assign to this one.

Public Member Functions

licenses

```
cmStd vector<ccCvlString> licenses(
  const ccCvlString& database = cmT("cogtool.dat")) const;
```

Returns a vector of the licenses available on either a specific board or all boards, depending on how this security information object was constructed.

Parameters

database The name of the file in which the set of available licenses is returned. This must be a *cogtool.dat* database file. It can be specified as either a fully qualified file name (such as "*C:\myrelease\cogtool.dat*"), or simply a file name located on the system path.

Throws

ccSecurityInfo::DatabaseError

The given database file cannot be found or read.

isTimeLimited `bool isTimeLimited() const;`

Returns true if the board used to construct this object is a time-limited security device, false otherwise.

Notes

Always returns false if this security information object is constructed with an argument of *kSystem*, or is default constructed.

isSoftwareLicense

`bool isSoftwareLicense() const;`

Returns true if the board used to construct this object represents a software license. Otherwise, returns false.

Notes

Always returns false if this object was constructed with *kSystem*

isActive `bool isActive() const;`

Returns true if the time-limited security device is currently active.

Throws

ccSecurityInfo::NotTimeLimited

The board used to construct this object is not a time-limited security device (**isTimeLimited()** is false).

Notes

This method applies only to security information objects constructed with a time-limited security device (those for which **isTimeLimited()** is true).

isExpired `bool isExpired() const;`

Returns true if the time-limited security device is expired.

Throws

ccSecurityInfo::NotTimeLimited

The board used to construct this object is not a time-limited security device (**isTimeLimited()** is false).

Notes

This method applies only to security information objects constructed with a time-limited security device (those for which **isTimeLimited()** is true).

daysRemaining `double daysRemaining() const;`

Returns the number of days remaining, full and fractional, before the time-limited security device expires.

Throws

ccSecurityInfo::NotTimeLimited

The board used to construct this object is not a time-limited security device (**isTimeLimited()** is false).

Notes

This method applies only to security information objects constructed with a time-limited security device (those for which **isTimeLimited()** is true).

ccSemaphore

```
#include <ch_cvl/threads.h>

class ccSemaphore: public cc_Resource;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A semaphore is a resource with a non-negative count. A thread can lock a semaphore as long as its count remains greater than 0. Each lock of a semaphore decrements the semaphore's count.

Constructors/Destructors

ccSemaphore

```
ccSemaphore(c_Int32 initialCount = 0,
  c_Int32 maximumCount = 1);
```

Constructs a **ccSemaphore** with specified initial and maximum counts

Parameters

initialCount The initial count for this **ccSemaphore**. An *initialCount* of 0 corresponds to a fully locked state; **unlock()** must be called before this **ccSemaphore** can be locked.

maximumCount The maximum number of times that this **ccSemaphore** can be locked.

Notes

initialCount must be greater than or equal to 0 and less than or equal to *maximumCount*. *maximumCount* must be greater than or equal to 1.

Public Member Functions

lock

```
bool lock(double timeout=HUGE_VAL);
```

Decrements the count of this **ccSemaphore**. If this function was able to decrement the count, it returns true. If the count is already at 0 and remained so for the duration of the specified timeout, this function returns false.

Parameters

timeout The number of seconds to block. If you specify *HUGE_VAL* for *timeout*, the thread will wait forever or until the semaphore becomes availab.e.

Throws

cc_Resource::BrokenLock
The **cc_Resource::breakLocks()** static function was invoked on this thread.

unlock `void unlock();`

Increments the count of this **ccSemaphore**. If the count is already at its maximum value, this function has no effect.

ccSensor

```
#include <ch_cvl/sensor.h>

class ccSensor;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This abstract class is used by other classes to provide functions that get information from temperature sensors.

Constructors/Destructors

This is an abstract class. Only derived classes can creates instances of this class.

Public Member Functions

temperatureSensorCpu

```
virtual ccTemperatureSensor temperatureSensorCpu() = 0;
```

In derived classes, returns a temperature sensor object that corresponds to the temperature sensor under the CPU.

temperatureSensorCvm

```
virtual ccTemperatureSensor temperatureSensorCvm() = 0;
```

In derived classes, returns a temperature sensor object that corresponds to the temperature sensor under the Cognex Video Module (CVM).

ccSensor

ccShape

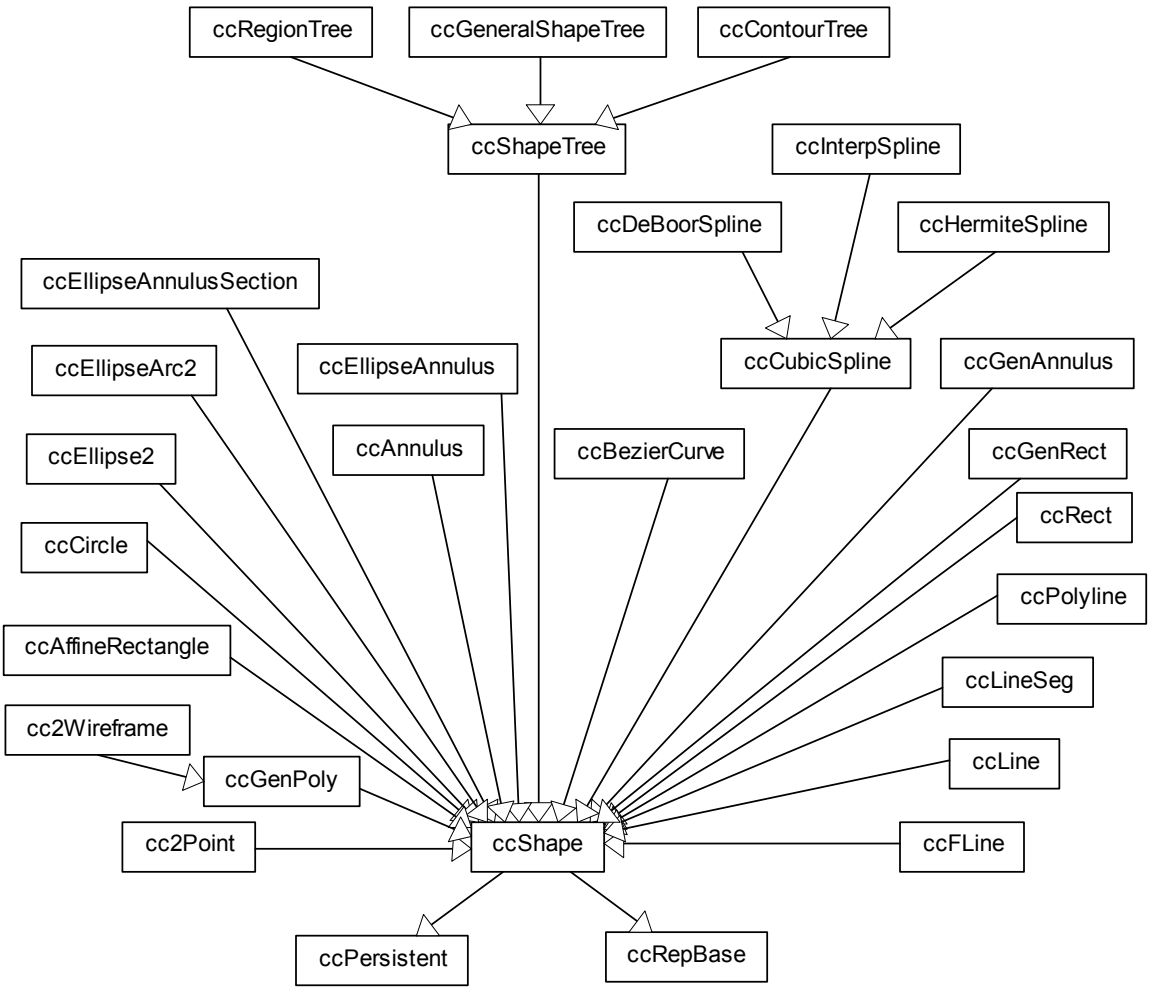
```
#include <ch_cvl/shapbase.h>

class ccShape : public virtual ccPersistent,
  public virtual ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccShape** class is the abstract base class from which most CVL shapes are derived. Exceptions are some simple shapes (such as **ccPoint, ccPointSet, ccCross**, and **ccCoordAxes**) and some deprecated shapes (such as **ccPolygon**).



The above figure shows the **ccShape** class inheritance hierarchy.

Constructors/Destructors

Notes

As **ccShape** is an abstract base class, your application will not instantiate it directly.

~ccShape `virtual ~ccShape();`

Destructor. Deletes this shape.

Enumerations

ClipResult `enum ClipResult;`

This enumeration is used to classify a clipping operation into one of four categories: inside, outside, enclose, or partial.

Value	Meaning
eInside	The shape to clip lies completely inside the clipping region.
eOutside	The shape to clip lies completely outside the clipping region.
eEnclose	The shape to clip lies outside the clipping region and encloses (surrounds) it. An enclose status can only occur when the shape is a closed curve. For example, a circle may completely enclose a rectangular clipping region. In this case, the result of the clipping operation is an empty shape, just as it would be if the shape were completely outside the clipping region but not surrounding it.
ePartial	The shape to clip lies partially inside and partially outside the clipping region.

Public Member Functions

clone `virtual ccShapePtrh clone() const = 0;`

Returns a pointer to a copy of this shape.

isOpenContour `virtual bool isOpenContour() const = 0;`

Returns true if and only if this shape is an open contour. A shape is an open contour if and only if it is a continuous curve with distinguishable start and end points, which may coincide, through which it can be connected to other open contours.

Notes

Most shapes are either always open (for example, **ccLineSegment**) or always closed (for example, **ccCircle** and **ccRect**). For shapes that can be open or closed, the open/closed status is usually independent of the actual geometry. For example, see the comments under the **ccPolyline** constructor.

isRegion

```
virtual bool isRegion() const = 0;
```

Returns true if and only if this shape is a region. A shape is a region if and only if it divides the plane into well-defined inside and outside regions. The inside must be finite. Empty shapes are not regions. A region shape may contain multiple contours (for example, **ccAnnulus**).

isFinite

```
virtual bool isFinite() const = 0;
```

Returns true if and only if this shape has finite extent. Empty shapes are considered to have finite extent. A **ccShape** has finite extent if it lies within a bounding box.

isEmpty

```
virtual bool isEmpty() const = 0;
```

Returns true if and only if the set of points that lie on the boundary of this shape is empty. For example, a **ccPolyline** with zero vertices is empty.

hasTangent

```
virtual bool hasTangent() const = 0;
```

Returns true if and only if the set of points that lie on the boundary of this shape is infinite, and there is a well-defined tangent at all but a finite number of these points, including the start point and end point, if any.

Notes

This function returns true for most shapes. Examples of exceptions are **cc2Points**, **ccLineSeg** objects of zero length, and empty shapes.

isDecomposed

```
virtual bool isDecomposed() const = 0;
```

Returns true if this shape is already decomposed. That is, if invoking **decompose()** on this shape would produce one that is identical in structure to the original. See **decompose()** for details.

isReversible

```
virtual bool isReversible() const = 0;
```

Returns true if and only if this shape can be reversed. See **reverse()** for details.

boundingBox `virtual ccRect boundingBox() const = 0;`

Returns the smallest rectangle that encloses this shape.

Throws

ccShapesError::InfiniteExtent
isFinite() returns false.

ccShapesError::EmptyShape
isEmpty() returns true.

Notes

This function succeeds the deprecated **encloseRect()**, defined for most shapes. The main differences are the throws and the fact that it is virtual.

nearestPoint `virtual cc2Vect nearestPoint(const cc2Vect &p) const = 0;`

Returns the nearest point on the boundary of this shape to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

p The point.

Notes

The returned result is exact unless otherwise noted.

Throws

ccShapesError::EmptyShape
isEmpty() returns true.

distanceToPoint `inline double distanceToPoint(const cc2Vect& p) const;`

Returns the minimum distance from this shape to the given point, computed as the distance between the point *p* and **nearestPoint(p)**.

Parameters

p The point.

Notes

The returned result is exact unless otherwise noted.

This function succeeds the deprecated **distToPoint()**, defined for most shapes. The main differences are the throws and the fact that this function is defined in terms of the virtual function **nearestPoint()**.

Throws

ccShapesError::EmptyShape
isEmpty() returns true.

perimeter

```
virtual double perimeter() const;
```

Returns the perimeter of this shape. This value is exact unless otherwise noted.

The perimeter is returned in the same units as those used for the shape itself. If the shape is defined in image coordinates, the units are pixels. If the shape is defined in client coordinates, the units are those used for the client coordinates.

Notes

If **hasTangent()** is false, this method returns zero, even if this shape is empty.

The base class implementation of this method may not be the most efficient for derived shapes.

Throws

ccShapesError::InfiniteExtent
isFinite() is false for this shape.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this shape to the given point, as determined by **nearestPoint()**.

Parameters

info Shape information for this shape. Must be valid for this particular shape. See **ccShapeInfo** for details.

point The point.

Throws

ccShapesError::EmptyShape
isEmpty() is true for this shape.

ccShapesError::BadParams
info is detectably invalid for this shape.

perimPoint

```
cc2Vect perimPoint(const ccShapeInfo& info,
  const ccPerimPos& pos) const;
```

Returns the point along the perimeter of this shape that the given perimeter position describes.

Parameters

info Shape information for this shape. Must be valid for this particular shape. See **ccShapeInfo** for details.

pos The perimeter position. Must represent a valid perimeter position for this shape.

Throws

ccShapesError::BadParams
Either *info* or *pos* is detectably invalid for this shape. This includes the case where this shape is empty, as an empty shape cannot have any valid perimeter positions.

perimPointAndTangent

```
void perimPointAndTangent(const ccShapeInfo& info,
  const ccPerimPos& pos, cc2Vect& point, ccRadian& angle)
  const;
```

Returns the point along the perimeter of this shape defined by the given perimeter position, and computes the tangent angle at that point.

Parameters

info Shape information for this shape. Must be valid for this particular shape. See **ccShapeInfo** for details.

pos The perimeter position.

point Return parameter for the point described by the perimeter position *pos*.

angle Return parameter for the tangent angle at *point*.

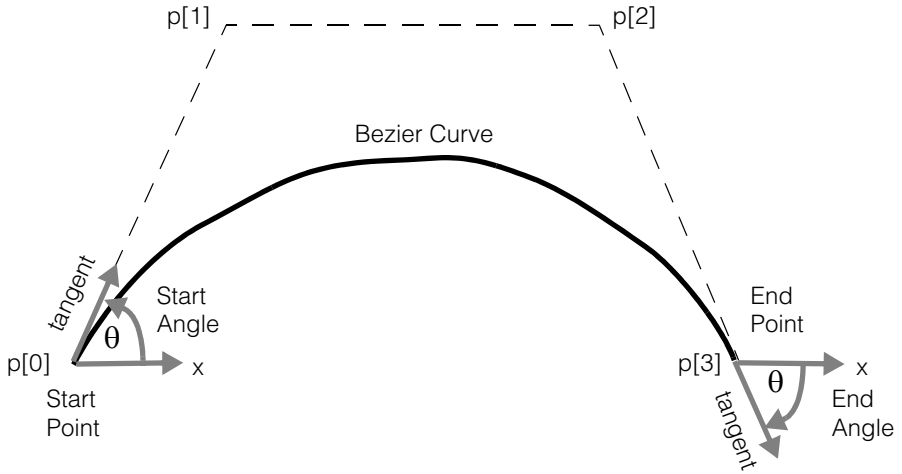
Throws

ccShapesError::BadParams
Either *info* or *pos* is detectably invalid for this shape. This includes the case where this shape is empty, as an empty shape cannot have any valid perimeter positions.

ccShapesError:NoTangent
There is no tangent at the calculated point along the perimeter of this shape.

startPoint `virtual cc2Vect startPoint() const;`

Returns the starting point of this open contour shape. For example, let p[0], p[1], p[2], and p[3] be the four control points of a Bezier curve. The starting point of the Bezier curve in the following figure is the point p[0].



Throws

ccShapesError::NotOpenContour
This shape is not an open contour.

Notes

This method is specific to open contour shapes. The default implementation, which throws *ccShapesError::NotOpenContour*, is overriddden only for derived shapes that are open contours.

endPoint `virtual cc2Vect endPoint() const;`

Returns the ending point of this open contour shape. The ending point of the Bezier curve in the figure above is the point p[3].

Throws

ccShapesError::NotOpenContour
This shape is not an open contour.

Notes

This method is specific to open contour shapes. The default implementation, which throws *ccShapesError::NotOpenContour*, is overriddden only for derived shapes that are open contours.

startAngle `virtual ccRadian startAngle() const;`

Returns the starting tangent direction of this open contour shape. The figure on the previous page shows an example of the starting angle for a Bezier curve.

Throws

ccShapesError::NotOpenContour
This shape is not an open contour.

ccShapesError::NoTangent
hasTangent() is false for this shape.

Notes

This method is specific to open contour shapes. The default implementation, which throws *ccShapesError::NotOpenContour*, is overriddden only for derived shapes that are open contours.

endAngle `virtual ccRadian endAngle() const;`

Returns the ending tangent direction of this open contour shape. The figure on the previous page shows an example of the ending angle for a Bezier curve.

Throws

ccShapesError::NotOpenContour
This shape is not an open contour.

ccShapesError::NoTangent
hasTangent() is false for this shape.

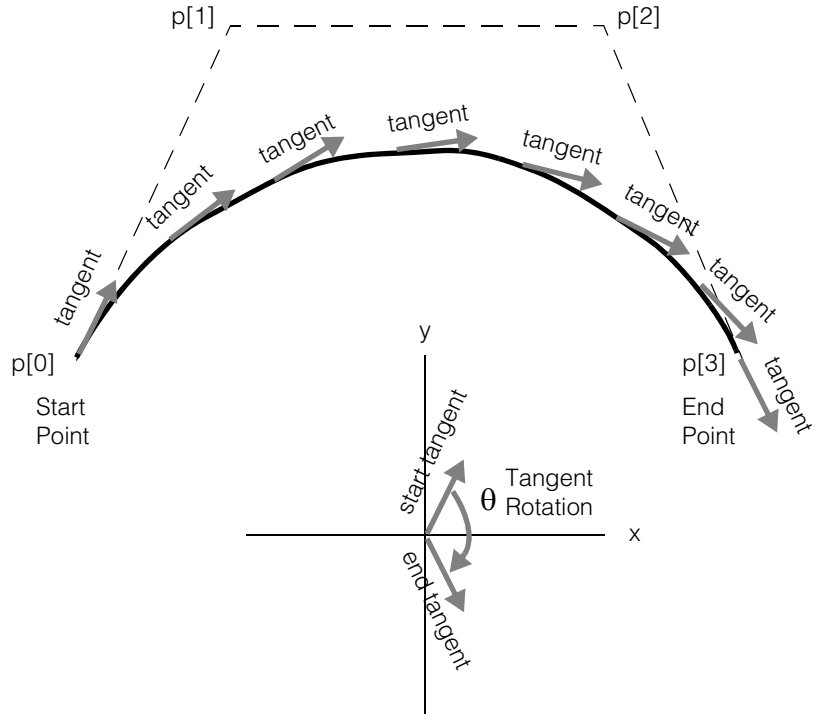
Notes

This method is specific to open contour shapes. The default implementation, which throws *ccShapesError::NotOpenContour*, is overriddden only for derived shapes that are open contours.

tangentRotation `virtual ccRadian tangentRotation() const;`

Returns the net signed angle through which the tangent vector rotates along the contour of this shape from the start point to the end point. This method is specific to open contour shapes.

The following figure shows an example of tangent rotation for a Bezier curve. In this example, the tangent rotation is approximately 120°.



Throws

ccShapesError::NotOpenContour
This shape is not an open contour.

ccShapesError::NoTangent
hasTangent() is false for this shape.

Notes

The tangent rotation is an integrated angle along the curve, theoretically equal to *endAngle() - startAngle() + 2k*π radians for some integer *k*.

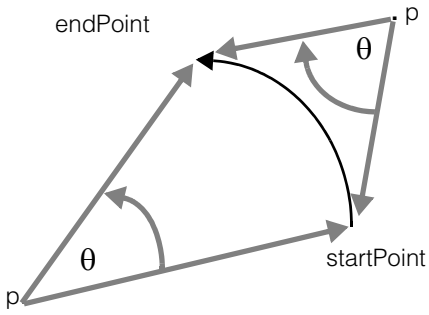
Tangent rotation is used to determine the handedness of a composite contour that includes this shape.

This method is specific to open contour shapes. The default implementation, which throws *ccShapesError::NotOpenContour*, is overriddden only for derived shapes that are open contours.

windingAngle `virtual ccRadian windingAngle(const cc2Vect &p) const;`

For an open contour shape, let *t* be a point that moves continuously along the curve from the start point to the end point. The winding angle is the net signed angle through which the vector *p->t* rotates as *t* traces the curve.

For example, in the following figure, when *p* is to the right of the ellipse arc, the winding angle θ is roughly $-\pi/4$ (or -45°). When p is to the left of the ellipse arc, the winding angle is roughly $+\pi/6$ (or $+30^\circ$).



Parameters

p The start point of the vector *p->t* whose angle is measured as the end point *t* traces the curve.

Throws

ccShapesError::NotOpenContour
This shape is not an open contour.

Notes

This method is used for determining whether *p* is inside or outside of a region whose boundary includes this shape.

This method is specific to open contour shapes. The default implementation, which throws *ccShapesError::NotOpenContour*, is overriddden only for derived shapes that are open contours.

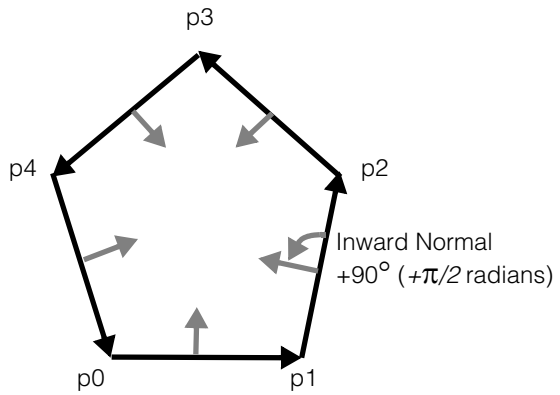
isRightHanded `virtual bool isRightHanded() const;`

Returns true if and only if this region shape is right-handed.

Handedness is an intrinsic property of a shape, and is independent of the coordinate system in which the shape is displayed. For non-intersecting, closed contours, the following criteria are all equivalent ways of characterizing right-handedness:

- The tangent vector rotates through $+2\pi$ radians along the curve.
- The inward normal is rotated $+\pi/2$ radians from the tangent vector.
- In an approximating polygon with vertices $p_1,\ldots, p_N$, the sum of $p_i \times p_{(i+1)}$ over all i is positive, where the x denotes vector cross product.

Right-Handed Shape



The criteria for left-handedness are obvious negations of the above criteria. The positive angle direction is the direction in which x rotates into y. The handedness criteria presume a direction of travel along the contour (a tangent vector). This is defined as the direction in which points are sampled by the method **sample()**, below. Handedness is not equivalent to clockwise/counterclockwise designations, as the latter are dependent on the coordinate system in which the shape is displayed. For a particular display coordinate system, however, there is a correspondence. For example, if the display coordinate system's x-axis points right and its y-axis points down, then a closed curve is right-handed if and only if it traces the enclosed area in a clockwise direction.

Throws

ccShapesError::NotRegion
This shape is not a region.

Notes

This method is specific to region shapes. Boundaries of region shapes are assumed not to self-intersect.

within

```
virtual bool within(const cc2Vect &p) const;
```

Returns true if and only if the given point is inside of the region defined by this shape.

Parameters

p The point.

Throws

ccShapesError::NotRegion
This shape is not a region.

Notes

This method is specific to region shapes. Boundaries of region shapes are assumed not to self-intersect.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const = 0;
```

Returns this shape mapped by X.

Parameters

X The transformation object.

Notes

The returned shape is of the same type as the original shape unless otherwise noted in a derived class.

reverse

```
virtual ccShapePtrh reverse() const;
```

Returns the reversed version of this shape. A reversed shape describes the same locus of points as the original, but parameterizes this locus of points in the opposite direction. Not all shapes are reversible. The **isReversible()** method indicates whether a shape can be reversed. All open contour shapes are reversible; for these shapes, **startPoint()** in the reversed shape corresponds to **endPoint()** in the original shape, and vice versa. Most other shapes (such as **ccCircle**, **ccRect**, and **ccGenRect**) are not reversible. Notable exceptions are closed polygon-like shapes (such as **ccGenPoly**, **ccPolyline**, and **ccCubicSpline**). These shapes can be reversed even when they are closed.

Notes

Sampling a reversed shape always generates sample points that traverse the shape in the opposite direction from that traversed by sampled points from the original shape.

Throws

ccShapesError::NotReversible
This shape is not reversible.

decompose

```
virtual ccShapePtrh decompose() const = 0;
```

Returns a new shape describing the same geometry as this shape, but using only a subset of shape primitives. Decomposed shapes contain only line segments, ellipse arcs, and Bezier curves as primitives, possibly arranged into shape hierarchies. A decomposed shape describes the same locus of points as the original shape, and contains the same connectivity and tangent direction information. For example, a **ccRect** is decomposed into four line segments grouped into a right-handed **ccContourTree**.

A **ccShape** can be decomposed into only the following shapes:

- Decomposed **ccGeneralShapeTree**s
- Decomposed **ccContourTree**s
- **ccLineSeg**s
- **ccEllipseArc**s
- **ccBezierCurve**s

ccGeneralShapeTree::decompose() returns a **ccGeneralShapeTree** that contains decomposed versions of any children that are not **ccGeneralShapeTree**s or **ccRegionTree**s, in addition to the decomposed children of any **ccGeneralShapeTree** or **ccRegionTree** children. A decomposed **ccGeneralShapeTree** can contain only the following types of children:

- Decomposed **ccContourTree**s
- **ccLineSeg**s
- **ccEllipseArc**s
- **ccBezierCurve**s

ccContourTree::decompose() returns a decomposed **ccContourTree** that is a merged version of its decomposed children. A decomposed **ccContourTree** can contain only the following types of children:

- **ccLineSeg**s
- **ccEllipseArc**s
- **ccBezierCurve**s

Notes

Decomposition is a many-to-one transformation. It is not possible to infer from the decomposition alone the structure of the original shape.

Shapes for which **isEmpty()** is true or **isFinite()** is false decompose into **ccGeneralShapeTree**s or **ccContourTree**s without any children. Points decompose into zero-length line segments with both endpoints coincident with the original point.

---

clip

```
ccShapePtrh clip(const ccPolyline &clipRegion,
  bool checkHandedness = true) const;

ccShapePtrh clip(const ccPolyline &clipRegion,
  ClipResult &clipResult, bool checkHandedness = true)
const;
```

- ```
  ccShapePtrh clip(const ccPolyline &clipRegion,
    bool checkHandedness = true) const;
  ```

  Returns the portion of this shape lying within the clipping region. The returned shape may not have the same types of component primitive shapes as this shape.

  Parameters

  clipRegion The clipping region. *clipRegion* must be a closed, convex, polyline. It must be a right-handed polyline if *checkHandedness* is false.

  checkHandedness
  If true, the handedness of the clip region is determined prior to clipping. If false, handedness is not determined prior to clipping. *clipRegion* must be a right-handed polyline if *checkHandedness* is false. The default is true.

- ```
  ccShapePtrh clip(const ccPolyline &clipRegion,
    ClipResult &clipResult, bool checkHandedness = true)
  const;
  ```

  Returns the portion of this shape lying within the clipping region, and a result object containing information on the type of clipping performed. The returned shape may not have the same types of component primitive shapes as this shape.

  Parameters

  clipRegion The clipping region. *clipRegion* must be a closed, convex, polyline. It must be a right-handed polyline if *checkHandedness* is false.

clipResult Result object produced by the clipping operation; contains information on the type of clipping performed. If this shape is empty, the returned *clipResult* is *eInside* (see enum **ClipResult**).

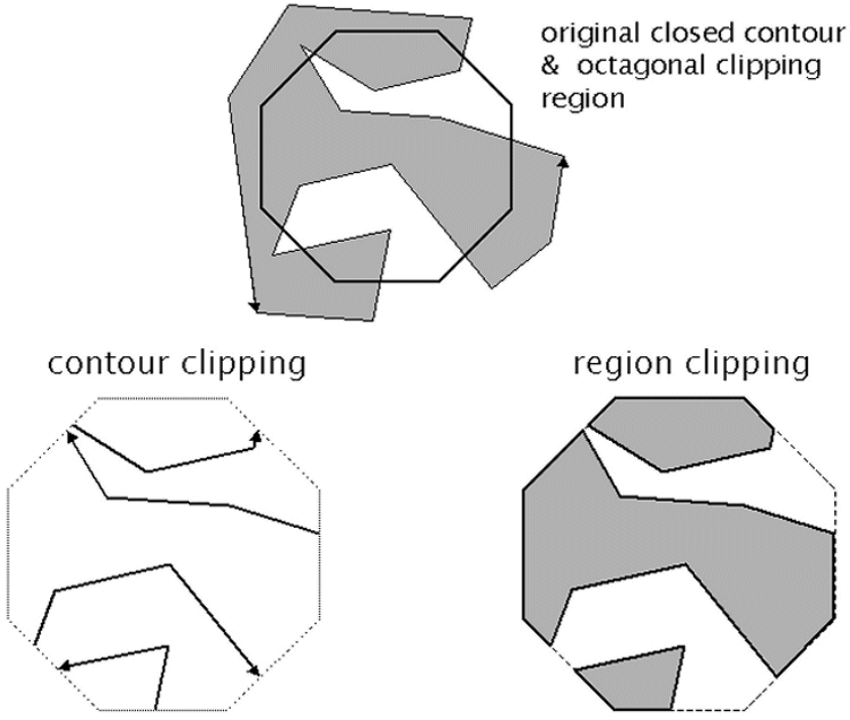
checkHandedness

If true, the handedness of the clip region is determined prior to clipping. If false, handedness is not determined prior to clipping. *clipRegion* must be a right-handed polyline if *checkHandedness* is false. The default is true.

Notes

If *checkHandedness* is true, the handedness of the clip region is determined prior to clipping. The clipping algorithms need this information. To avoid the overhead of automatically determining handedness, pass in a right-handed clip region, and a value of false for *checkHandedness*.

Closed shapes are generally clipped as boundaries, not as regions. The area they enclose is not considered part of the shape. Thus, a partially clipped region shape is not a region, but rather a collection of individual boundary segments. The class **ccRegionTree** is the notable exception to this rule.



subShape

```
virtual ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this shape over the given perimeter range. The perimeter of the final returned shape is equal to the absolute value of the distance component of *range*, assuming the distance is not clipped (see notes).

Parameters

info Shape information for this shape. Must be valid for this particular shape.

range The perimeter range. The perimeter position component of this value must be valid for this particular shape.

Notes

The returned shape does not extend beyond the contour containing the reference position. This method never jumps gaps between contours.

If the contour containing the reference position of *range* is a finite open contour, the distance is clipped internally, if necessary to generate a position on the same contour. If clipping occurs, the returned subshape extends to the start point or the end point of the contour.

If the contour containing the reference position of *range* is a finite closed contour of perimeter *p*, the distance *d* is clipped internally to the range *-p* through *+p*, inclusive. This generates a subshape with a maximum perimeter of *p*. If clipping occurs, the returned subsape is closed, otherwise it is open. Distances wrap around the branch cut of a closed contour appropriately.

When *d* is negative, the tangent direction along the returned shape is opposite to that of the original. As with **reverse()**, this can cause a reversal in the effective model properites. See the *Shape Model Properties* section of the *Shape Models* chapter of the *CVL Vision Tools Guide* for more information on the effects of shape manipulation on shape model properties.

Throws

ccShapesError::BadParams

Either *info* or the perimeter position component of *range* is detectably invalid for this shape. This includes the case where this shape is empty, as an empty shape cannot have any valid perimeter positions.

sample

```
virtual void sample(const ccSampleParams &params,
  ccSampleResult &result) const = 0;
```

Returns sample positions, and possibly tangents, along this shape. A single contour, open or closed, can be modeled by a chain (or sequence) of positions along the contour. Polygonal contours are modeled exactly in this way. Curved contours are only

approximated as the position chain contains a finite number of elements. Optionally, a chain of tangent directions, one corresponding to each sampled position, can also be computed. The **ccSampleParams** object passed in specifies details of how the sampling should be done. The **ccSampleResult** object returned contains the results of the sampling operation.

This method appends position and tangent chains corresponding to this shape to the supplied **ccSampleResult** object. Each chain corresponds to one contiguous portion of the shape. For example, most primitive shapes add one position and tangent chain. Primitive annulus shapes add two position and tangent chains. Hierarchical shapes may add an arbitrary number of chains.

Tangent direction samples may or may not be computed along with position samples, as specified by the **computeTangents()** flag of the supplied **ccSampleParams** object. Once samples have been stored into a **ccSampleResult** object, it is an error to change this flag on subsequent sampling operations. Attempting to do so will result in a throw. This is necessary to maintain a one-to-one correspondence between positions and tangents if tangents are computed.

Parameters

params Parameters object specifying details of how the sampling should be done.

result Result object to which position and tangent chains are stored.

Throws

ccShapesError::SampleOverflow
The supplied spacing and tolerance bounds require more than **ccSampleParams::maxPoints()** samples to be generated along a single element of a primitive shape (see **ccSampleParams**).

ccShapesError::InfiniteExtent
This shape does not have finite extent.

ccShapesError::BadParams
Position samples without tangent directions have already been stored into *result*, but *params* specifies that tangents should be computed. Conversely, tangent direction samples have already been stored into *result*, but *params* specifies that tangents should not be computed.

Notes

The **sample()** method does not repeat vertices for closed primitive contours, such as **ccCircles**. See **ccSampleParams::duplicateCorners()** for more information.

Overrides of this function must return positions that are on the shape. For example, **ccCircle::sample()** should return vertices of an inscribed polygon, not a circumscribed one. Certain algorithms that call **sample()** assume this behavior.

Primitive annulus shapes and region trees generate position chains that bound the inside of the shape in a consistent manner. The chain around the outside, therefore, is in the opposite direction to the chain around the inside hole.

If *params* specifies that tangents should be computed, then this function ignores shapes for which **hasTangent()** is false, such as **cc2Point**. Such shapes do not contribute any samples to the position or tangent chains stored in *result*. If *params* specifies that tangents should not be computed, then shapes such as **cc2Point** will still generate position samples.

Typedefs

ccShapePtrh

```
typedef ccPtrHandle<ccShape> ccShapePtrh;
```

Pointer handle to a **ccShape**.

ccShapePtrh_const

```
typedef ccPtrhHandle_const<ccShape> ccShapePtrh_const;
```

Pointer handle to a `const` **ccShape**.

ccShapeInfo

```
#include <ch_cvl/shapbase.h>

class ccShapeInfo;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class encapsulates detailed information about a shape. The perimeter and parameterization functions of **ccShape** require that this information be explicitly precalculated. This ensures that repeated calls to those functions on an unchanging shape are maximally efficient.

A **ccShapeInfo** object can be constructed for any primitive shape, finite or infinite, but not for a non-primitive shape. For example, a **ccShapeInfo** object can be constructed for a line, but not for a general shape tree containing a line. You can convert an infinite shape into a finite one by clipping it.

Similarly, shape operations that require information stored in a **ccShapeInfo** object can be performed on infinite primitive shapes, but not on infinite non-primitive shapes.

A **ccShapeInfo** object contains information for a particular shape. It is not valid for any other shape, whether obtained from the original through mapping, reversing, clipping, decomposing, or any other type of modification.

Constructors/Destructors

ccShapeInfo

```
ccShapeInfo(const ccShape& shape);
```

Constructs a **ccShapeInfo** object for the given shape.

Parameters

shape The shape.

Throws

ccShapesError::InfiniteShape
The shape this **ccShapeInfo** object describes is infinite and is not a primitive. If the described shape is infinite but primitive, no exception is thrown.

Public Member Functions

perimeter

```
double perimeter() const;
```

Returns the perimeter of the shape that this **ccShapeInfo** object describes.

Throws

ccShapesError::InfiniteExtent
The shape that this **ccShapeInfo** describes is not finite.

Notes

This function is never slower than **ccShape::perimeter()** and may be much faster.

See **ccShape::perimeter()** for more information.

---

perimPos

```
ccPerimPos perimPos(double distanceAlong) const;

ccPerimPos perimPos(const ccPerimPos &referencePos,
  double distance) const;
```

- `ccPerimPos perimPos(double distanceAlong) const;`

Returns the position at the given distance along the perimeter of the shape that this **ccShapeInfo** object describes.

Parameters

distanceAlong The distance from the starting point along the perimeter of the shape.

Throws

ccShapesError::EmptyShape
This shape is empty.

Notes

If the shape that this **ccShapeInfo** object describes is finite, **distanceAlong()** is automatically clipped to the range 0 through **perimeter()**, inclusive. For infinite primitive shapes, **distanceAlong()** is meaningful over all real values, signed and unsigned, and is not clipped.

- `ccPerimPos perimPos(const ccPerimPos &referencePos, double distance) const;`

Returns the position at the given signed distance along the perimeter of the shape that this **ccShapeInfo** object describes, relative to the given reference position. This overload always returns a perimeter position on the same contour as the supplied reference position.

Parameters

referencePos The reference position. Must be a valid perimeter position for the exact shape that this **ccShapeInfo** object describes.

distance The distance from the reference position.

Notes

If the contour containing the reference position is a finite open contour, the distance is clipped internally, if necessary, to generate a position on the same contour. If clipping occurs, the returned position corresponds to either the start point or the end point of the contour.

If the contour containing the reference position is a finite closed contour of perimeter *p*, the distance is clipped internally to the range *-p* through *+p*, inclusive. This generates a new position on the same contour.

Distances wrap around the branch cut of a closed contour appropriately. Hence, it is possible, for example, for the returned position to have a smaller **distanceAlong()** value than the reference position, even when *distance* is positive.

Contours are not synonymous with primitives. Some contours (for example, contour trees) comprise multiple primitives, and some primitives (for example, annulus shapes) comprise multiple contours. Within contour trees, the returned position may lie on a different primitive than the reference position. Within annulus primitives, positions on the contour not containing the reference position are inaccessible from this function.

Throws

ccShapesError::BadParams

referencePos is detectably invalid for this **ccShapeInfo** object.

distanceAlong `double distanceAlong(const ccPerimPos& pos) const;`

Returns the distance from the given point along the perimeter of the shape that this **ccShapeInfo** describes.

Parameters

pos The point. Must be a valid perimeter position for the exact shape that this **ccShapeInfo** object describes.

Notes

If the shape that this **ccShapeInfo** describes is finite, the returned value lies in the range 0 through **perimeter()**, inclusive. If this **ccShapeInfo** object describes an infinite primitive shape, the returned value is unconstrained, and may be negative.

Throws

ccShapesError::BadParams

pos is detectably invalid for this **ccShapeInfo** object.

ccShapeMaskValue

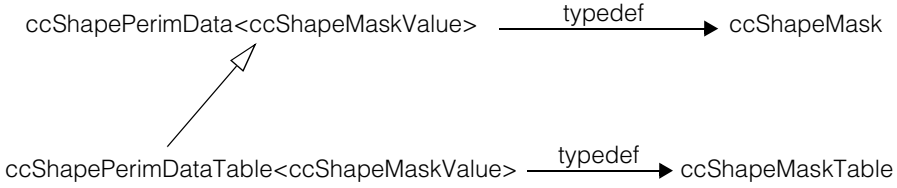
```
#include <ch_cvl/shapemsk.h>

class ccShapeMaskValue
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class holds mask values suitable for representing masking information for CVL shape descriptions (**ccShape** objects). **ccShapeMaskValue** is used to instantiate the templated class **ccShapePerimData** to create **ccShapeMask** and **ccShapeMaskTable** which are used by the **ccBoundaryInspector** tool to mask out unwanted parts of a **ccShape** model. See the following diagram:



Constructors/Destructors

ccShapeMaskValue

```
ccShapeMaskValue(
  ShapeMaskValue maskValue =
                        ShapeMaskValue::kDefaultMaskValue);
```

Constructs a **ccShapeMaskValue** object with the given mask value.

Parameters

maskValue The shape mask value.

Operators

operator==

```
bool operator== (const ccShapeMaskValue &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

operator!=

```
bool operator!= (const ccShapeMaskValue &that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that	The object to compare with this object.

Enumerations

ShapeMaskValue

```
enum ShapeMaskValue
```

Defines the allowed shape mask values.

Value	Meaning
eNotMasked = 0	Not masked (will be processed by the tool)
eMasked = 1	Masked (masked areas are ignored)
kDefaultMaskValue = *eNotMasked*	The default

Public Member Functions

maskValue

```
ccShapeMaskValue::ShapeMaskValue maskValue() const;
void maskValue(ccShapeMaskValue::ShapeMaskValue maskVal);
```

The shape mask value. Must be one of the **ShapeMaskValue** enums.

- `ccShapeMaskValue::ShapeMaskValue maskValue() const;`

  Returns the current mask value.

- `void maskValue(ccShapeMaskValue::ShapeMaskValue maskVal);`

  Sets a new mask value.

  Parameters

  maskVal The new mask value.

ccShapeMaskValue

ccShapeModel

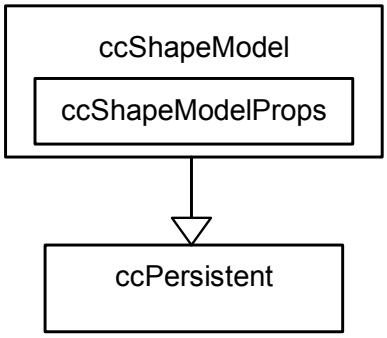
```
#include <ch_cvl/shapemod.h>

class ccShapeModel : public virtual ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccShapeModel** class is a *mixin* class that provides shape model properties for a shape. It also provides several static functions that help to set polarity for shapes.



The above figure shows the **ccShapeModel** class inheritance hierarchy.

ccShapeModel inherits from **ccPersistent**. It contains an instance of **ccShapeModelProps**, which specifies weight and polarity properties for a shape.

See the *Shape Models* chapter of the *CVL Vision Tools Guide* for more information on shape models.

Constructors/Destructors

ccShapeModel

```
ccShapeModel();
ccShapeModel(const ccShapeModelProps& props);
virtual ~ccShapeModel();
```

- `ccShapeModel();`

  Default constructor. Constructs a **ccShapeModel** object with default properties (see **ccShapeModelProps** on page 2937 for details).

- `ccShapeModel(const ccShapeModelProps& props);`

  Constructs a **ccShapeModel** object using the supplied shape model properties.

  Parameters

  props The shape model properties.

- `virtual ~ccShapeModel();`

  Destructor. Destroys this **ccShapeModel** object.

Operators

operator==

```
bool operator==(const ccShapeModel& other) const;
```

Returns true if this **ccShapeModel** is exactly equal to *other*, and false otherwise.

Parameters

other The other **ccShapeModel** object.

operator!=

```
bool operator!=(const ccShapeModel& other) const;
```

Returns true if this **ccShapeModel** is not exactly equal to *other*, and false otherwise.

Parameters

other The other **ccShapeModel** object.

Public Member Functions

modelProps

```
const ccShapeModelProps& modelProps() const;
void modelProps(const ccShapeModelProps& props);
```

- `const ccShapeModelProps& modelProps() const;`

  Gets the shape model properties.

- `void modelProps(const ccShapeModelProps& props);`

  Sets the shape model properties.

  Parameters

  props The shape model properties.

Static Functions

shapeModelProps

```
static ccShapeModelProps shapeModelProps(
  const ccShape& shape);
```

Returns the model properties (explicit or implicit) for the specified shape.

Parameters

shape The shape.

Notes

These properties are not necessarily the effective model properties, as they do not account for the model properties for any existing ancestors of the specified shape.

setInsidePolarity

```
static ccShapePtrh setInsidePolarity(const ccShape& shape,
  bool isPositive = true);
```

Returns a new shape that is identical to the supplied shape except that the polarity of all regions is such that the insides are all positive if *isPositive* is true, and negative if it is false.

Parameters

shape The supplied shape.

isPositive If true (the default), the inside polarity of all regions of the returned shape is positive; otherwise, it is negative.

Notes

The polarities of non-region shapes are left unchanged.

The returned shape is gauranteed to be of type **ccShapeModel**.

Ignore polarity flags do not affect the behavior of this function.

This function requires that no regions of the supplied shape self-intersect.

isInsidePolarity

```
static bool isInsidePolarity(const ccShape& shape,
  bool isPositive);
```

Returns true in the following cases, and false otherwise:

- *isPositive* is true and the inside polarity of all regions of the supplied shape is positive.
- *isPositive* is false and the inside polarity of all regions of the supplied shape is negative.

Parameters

shape	The shape.
isPositive	See above.

Notes

This function always returns true if *shape* is a primitive shape that is not a region.

Ignore polarity flags do not affect the behavior of this function.

This function requires that no regions of this shape self-intersect.

areAnyPolaritiesIgnored

```
static bool areAnyPolaritiesIgnored(const ccShape& shape);
```

Returns true if the ignore polarity flag is true for any portion of the supplied shape, otherwise false.

Parameters

shape	The shape.

features

```
static ccFeatureletChainSetPtrh features(
  const ccShape& shape,
  const ccShapeModel::ccFeatureParams &params);
```

Returns featurelet chains approximating the boundaries of the supplied shape. This method internally calls **ccShape::sample()** on the supplied shape with the supplied sampling parameters, and then converts the sample result into a featurelet chain set according to the following algorithm:

1. Create a *featurelet chain set* for each chain in the sample result (between breaks). The featurelet chain is closed if and only if the corresponding sample chain is closed.
2. Set the *position* of each featurelet to the corresponding sample position.
3. If the sampling parameter **computeTangents()** is true, define the *angle* for each featurelet as the normal to the tangent angle of the corresponding sample point on the positive side of the shape. The positive side is determined by the effective polarity of that portion of the shape from which the feature was sampled. An exception to this rule applies if **duplicateCorners()** is false (see note below for more information). If **computeTangents()** is false, the featurelet angles are undefined.
4. Set the *isMod180* flag for each featurelet to true if and only if the effective ignore polarity flag for that portion of the shape from which the feature was sampled is true.
5. Determine the *weight* of each featurelet. The weight of a featurelet is the product of the effective weight along that portion of the shape from which the feature was sampled and the weight scale supplied by *params*.
6. Determine the *magnitude* of each featurelet. The magnitude of a featurelet is the product of the effective magnitude along that portion of the shape from which the feature was sampled and the magnitude scale supplied by *params*.

Parameters

shape	The shape.
params	The parameters used to extract features from *shape*. These include both shape model specific parameters and shape sampling parameters.

Throws

See **ccShape::sample()** for possible exceptions that can be thrown from a sampling operation.

Notes

If **params.sampleParams().duplicateCorners()** is false, each feature located at a corner between contour tree children with inconsistent effective shape model properties represents the average of the two features that would have been generated at that corner had **duplicateCorners()** been true. In other words, the featurelet angle is the average angle computed after the polarity reversals,

magnitudes, and weights of the two children have been taken into account. The magnitude of the resulting feature is thus the average of the magnitudes of the two contour trees at that corner. The same is true for the weight. The *isMod180* flag of the resulting feaure is the logical OR of the *isMod180* flags of the two contour trees.

draw

```
static void draw(ccGraphicList& gList,
  const ccShape& shape,
  const ccGraphicProps& posGraphicProps,
  const ccGraphicProps& negGraphicProps,
  const ccGraphicProps& zeroGraphicProps = ccGraphicProps()
);
```

Draws the supplied shape in the supplied graphics list. Draws shapes with positive, negative, and zero effective weights using *posGraphicProps*, *negGraphicProps*, and *zeroGraphicProps*, respectively. Contours for which **ccGraphicProps::penColor()** is *passColor* are not drawn at all.

Polarity arrow heads are drawn for a contour if and only if the contour is drawn and **ccGraphicProps::arrowHead()** is true in the associated graphics properties for that contour. If arrowheads are drawn, '+' and '-' signs are placed on each side to indicate the effective polarity. If the effective ignore polarity flag is true, slashes are drawn through these signs.

Parameters

gList The graphics list.

shape The shape.

posGraphicProps
Graphic properties for shapes with positive effective weight.

negGraphicProps
Graphic properties for shapes with negative effective weight.

zeroGraphicProps
Graphic properties for shapes with zero effective weight. If no properties are supplied, the default **ccGraphicProps** drawing properties are used.

Notes

Phantom holes at the roots of region tree shapes are not drawn.

Shapes contained within the supplied shape for which no associated shape model type exists are not drawn.

record

```
static void record(ccDiagObject& obj,
  const ccPelBuffer_const<c_UInt8>& pelBuf,
  const ccShape& shape,
  const ccGraphicProps& posGraphicProps,
  const ccGraphicProps& negGraphicProps,
  const ccGraphicProps& zeroGraphicProps,
  const ccCvlString& annotation);
```

Records the supplied image and shape into the supplied diagnostic record. The shape is recorded in the client coordinate system of the image. Portions of the shape with positive, negative, and zero effective weights are recorded using *posGraphicProps*, *negGraphicProps*, and *zeroGraphicProps*, respectively. Contours for which **ccGraphicProps::penColor()** is *passColor* are not drawn at all.

Polarity arrow heads are recorded for a contour if and only if the contour is drawn and **ccGraphicProps::arrowHead()** is true in the associated graphics properties for that contour. If arrow heads are drawn, '+' and '-' signs are placed on either side to indicate the effective polarity. If the effective ignore polarity flag is true, slashes are drawn through these signs.

Tree structures within the shape are explicitly recorded as distinct levels within the record. All shape properties are recorded as text.

Parameters

obj The diagnostic record.

pelBuf The image.

shape The shape.

posGraphicProps
Graphic properties for shapes with positive effective weight.

negGraphicProps
Graphic properties for shapes with negative effective weight.

zeroGraphicProps
Graphic properties for shapes with zero effective weight. If no properties are supplied, the default **ccGraphicProps** drawing properties are used.

annotation A string that you can add to the diagnostic record.

Notes

Polarity graphics are not drawn for *phantom* holes at the roots of any region tree shapes within the supplied shape.

Shapes contained within the supplied shape for which no associated model shape type exists are not drawn.

rawShape

```
static ccShapePtrh rawShape(const ccShape &shape);
```

Returns a copy of the **ccShape** portion of the supplied shape stripped of any model properties.

Notes

The shape returned by this method is not the same as a reference to the shape portion of a shape model, which can be dynamically cast to a shape model. The shape returned by this method can neither be cast to a shape model, nor does it include the extra storage required for model properties.

Typedefs

ccShapeModelPtrh

```
typedef ccPtrHandle<ccShapeModel> ccShapeModelPtrh;
```

ccShapeModelPtrh_const

```
typedef ccPtrHandle_const<ccShapeModel>
ccShapeModelPtrh_const;
```

Deprecated Members

sampleWithPolarity

```
static void sampleWithPolarity(const ccShape& shape,
  const ccShape::ccSampleParams &params,
  ccShape::ccSampleResult &result);
```

This method is deprecated. Use **features()** instead.

ccShapeModelProps

```
#include <ch_cvl/shapemod.h>

class ccShapeModelProps;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccShapeModelProps** class encapsulates non-geometric information that is often required to describe the appearance of an object in an image. This information includes the following:

- *Polarity reversal flag*: if true, the default polarity of the shape is reversed (default = false).
- *Ignore polarity flag*: if true, polarity information should be ignored (default = false).
- *Weight*: relative importance of the shape as compared to other shapes (default = 1.0).
- *Magnitude*: strength of the boundary modeled by the shape. Magnitude can be any non-negative value (default = 1.0).

If these properties are not explicitly specified for a particular shape, the shape automatically inherits its properties from its parent if it is part of a tree. Otherwise it assumes the default values listed above.

Polarity

Polarity refers to the gradient direction for the boundary modelled by the shape. That is, the polarity of a shape specifies which side of its boundary is relatively dark with respect to the other. To encode this information, each shape has an unique *effective polarity*. The side of the shape boundary that is in the +90 degree direction from the tangent direction has one polarity (either *positive* or *negative*), and the side in the -90 degree direction from the tangent direction has the opposite polarity. The *polarity reversal flag* specifies whether the polarity for a shape is reversed from its normal polarity.

Ignore Polarity Flag

The *ignore polarity flag* specifies whether polarity information should be ignored for a shape. You can use this flag, for example, to effectively invalidate polarity information if it is unknown, or simply to ignore polarity information in cases where it is known. The ignore polarity flag is false by default, which means that the polarities of all shapes are valid and that vision tools can use them.

Weight

The *weight* property determines the relative importance of a shape. In the context of a shape tree, the weight for a child determines the importance of the feature that the child is modelling relative to the overall object that the shape tree is modelling. This assignment of relative importance is often dependent on the application. For example, when using a shape model to perform an alignment (for example, using PatMax), the weight determines how much this shape will contribute to the alignment score. The effective weight for a shape is assumed to be 1.0 by default, if not explicitly specified for that shape or by any of its ancestors.

Magnitude

The *magnitude* property specifies the non-negative strength of the boundary represented by a shape. Magnitude may be used, for example, to denote the edge contrast of an object. Magnitude is 1.0 by default.

Note See the *Shape Models* chapter of the *CVL Vision Tools Guide* for a more detailed discussion of shape model properties.

Constructors/Destructors

ccShapeModelProps

```
explicit
ccShapeModelProps(bool isReversedPolarity = false,
  double weight = 1.0,
  bool isIgnoredPolarity = false,
  double magnitude = 1.0);
```

Constructs a shape model properties object using the specified parameters. This constructor is used for explicit construction only and not for implicit conversions.

Parameters

isReversedPolarity
: If true, the polarity of the shape is reversed (default = false).

weight
: Relative importance of the shape (default = 1.0).

isIgnoredPolarity
: If true, polarity information for a shape should be ignored (default = false).

magnitude Strength of the boundary described by the shape (default = 1.0).

Throws

ccShapeModelDefs::BadParams
magnitude is less than 0.0.

Operators

operator==

```
bool operator==(const ccShapeModelProps& other) const;
```

Returns true if this object is exactly equal to *other*, and false otherwise.

Parameters

other The other **ccShapeModelProps** object.

operator!=

```
bool operator!=(const ccShapeModelProps& other) const;
```

Returns true if this object is not exactly equal to *other*, and false otherwise.

Parameters

other The other **ccShapeModelProps** object.

Public Member Functions

isReversedPolarity

```
void isReversedPolarity(bool b);
bool isReversedPolarity() const;
```

- `void isReversedPolarity(bool b);`

  Sets the *reversePolarity* flag.

  Parameters

  b True sets the polarity of the shape to reversed.

- `bool isReversedPolarity() const;`

  Gets the *reversePolarity* flag.

isIgnoredPolarity

```
void isIgnoredPolarity(bool b);
bool isIgnoredPolarity() const;
```

- `void isIgnoredPolarity(bool b);`

  Sets the ignore polarity flag.

  Parameters

  b — If true, polarity information for the shape is invalid and vision tools should ignore it.

- `bool isIgnoredPolarity() const;`

  Retrieves the ignore polarity flag.

magnitude

```
void magnitude(double m);
double magnitude() const;
```

- `void magnitude(double m);`

  Sets the magnitude property.

  Parameters

  m — The magnitude.

  Throws

  ccShapeModelDefs::BadParams
  m is less than 0.0.

- `double magnitude() const;`

  Gets the magnitude property.

weight

```
void weight(double w);
double weight() const;
```

- `void weight(double w);`

  Sets the weight property.

Parameters

w The weight.

- `double weight() const;`

  Gets the weight.

assign `ccShapePtrh assign(const ccShape& shape) const;`

Returns a new shape that is a shape model version of the supplied shape, containing these shape model properties. The returned shape is also of type **ccShapeModel**.

Parameters

shape The shape.

Notes

The safest way to convert any shape to an equivalent shape model is using **ccShapeModel::shapeModelProps(shape).assign(shape)**. This assigns the original model properties of shape, rather than default model properties, to the shape model. A case in point is **cc2Wireframe**s, which contain polarity information, and therefore possibly non-default model properties, even when they are not shape models.

combine

```
ccShapeModelProps combine(const ccShapeModelProps& other)
  const;
```

Returns a new shape model properties object that is the effective combination of these properties and the properties supplied by another properties object. Properties are combined such that:

- The resulting *weight* is the product of the two weights of this and the supplied properties object. If both values are negative, the resulting weight is zero.
- The resulting *polarity reversal flag* is the logical exclusive OR of the reversal flags of this and the supplied properties object.
- The resulting *ignore polarity flag* is the logical OR of the ignore flags of this and the supplied properties object.
- The resulting *magnitude* is the product of the magnitudes of this and the supplied properties object.

Parameters

other The other **ccShapeModelProps** object.

ccShapeModelProps

ccShapeModelTemplate<>

```
#include <ch_cvl/shapemod.h>

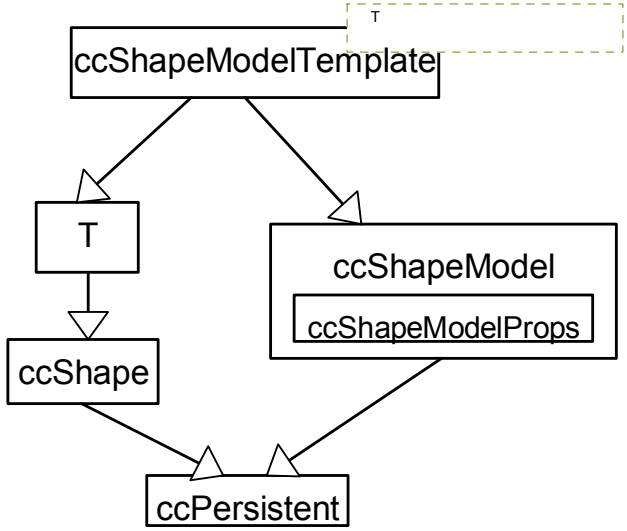
template <class P>
class ccShapeModelTemplate : public T,
  virtual public ccShapeModel;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccShapeModelTemplate<>** class endows a shape with shape model properties. It inherits the shape interface from the specified shape type, and the shape model properties from the **ccShapeModel** mixin class.

Note The shape type, **T**, must derive from **ccShape**.



The above figure shows the **ccShapeModelTemplate<>** class inheritance hierarchy.

See the *Shape Models* chapter of the *CVL Vision Tools Guide* for more information on the use of the **ccShapeModelTemplate<>** class.

Constructors/Destructors

ccShapeModelTemplate

```
ccShapeModelTemplate();

ccShapeModelTemplate(const ccShapeModelProps& props);

ccShapeModelTemplate(const T& shape,
  const ccShapeModelProps& props) : T(shape),
  ccShapeModel(props);
```

- `ccShapeModelTemplate();`

  Default constructor.

- `ccShapeModelTemplate(const ccShapeModelProps& props);`

  Constructs a shape model using the supplied shape model properties.

  Parameters

  props The shape model properties.

- ```
  ccShapeModelTemplate(const T& shape,
    const ccShapeModelProps& props) : T(shape),
    ccShapeModel(props);
  ```

  Constructs a shape model using the supplied shape and shape model properties.

  Parameters

  shape The shape.

  props The shape model properties.

Operators

operator==

```
bool operator==(const ccShapeModelTemplate<T>& other)
  const;
```

Returns true if this object is exactly equal to *other*, and false otherwise.

Parameters

other The other shape model object.
```

operator!=

```
bool operator!=(const ccShapeModelTemplate<T>& other)
  const;
```

Returns true if this object is not exactly equal to *other*, and false otherwise.

Parameters

other The other shape model object.

Public Member Functions

clone

```
ccShapePtrh clone() const;
```

Returns a pointer to a copy of this shape model.

reverse

```
ccShapePtrh reverse() const;
```

Returns the reversed version of this shape model object. See **ccShape::reverse()** for more information.

mapShape

```
ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this shape model object mapped by *X*.

Parameters

X The transformation object.

Throws

ccShapeModelDefs::NotSupported
This tree contains any shapes that do not have an associated model shape type.

See **ccShape::mapShape()** for more information.

decompose

```
ccShapePtrh decompose() const;
```

Returns a **ccContourTree** that describes the same geometry as this shape model, but using only a subset of primitive shapes. See **ccShape::decompose()** for more information.

subShape

```
ccShapePtrh subShape(const ccShapeInfo &info,
  const ccPerimRange &range) const;
```

Returns a pointer handle to the shape describing the portion of this shape model over the given perimeter range. The perimeter of the final returned shape is equal to the absolute value of the distance component of *range*, assuming the distance is not clipped.

Parameters

info	Shape information for this shape model.
range	The perimeter range.

Typedefs

There are several type definitions for **ccShapeModelTemplate<>** instantiations for specific shapes. For example, **ccCircleModel** is a type definition for a **ccShapeModelTemplate<>** that is instantiated with a **ccCircle** argument. A **ccCircleModel**, therefore, is a **ccCircle** shape with explicitly defined polarity and weight values. A **ccCircleModel** can use the polarity and weight values inherited from the **ccShapeModelProps** attribute of its **ccShapeModel** base class, or it can override them.

The following are the type definitions for **ccShapeModelTemplate<>** instantiations that CVL provides for all shapes.

ccGeneralShapeTreeModel

```
typedef ccShapeModelTemplate<ccGeneralShapeTree>
  ccGeneralShapeTreeModel;
```

ccContourTreeModel

```
typedef ccShapeModelTemplate<ccContourTree>
  ccContourTreeModel;
```

ccRegionTreeModel

```
typedef ccShapeModelTemplate<ccRegionTree>
  ccRegionTreeModel;
```

cc2PointModel

```
typedef ccShapeModelTemplate<cc2Point> cc2PointModel;
```

ccFLineModel

```
typedef ccShapeModelTemplate<ccFLine> ccFLineModel;
```

ccLineModel

```
typedef ccShapeModelTemplate<ccLine> ccLineModel;
```

ccLineSegModel

```
typedef ccShapeModelTemplate<ccLineSeg> ccLineSegModel;
```

ccRectModel

```
typedef ccShapeModelTemplate<ccRect> ccRectModel;
```

ccCircleModel

```
typedef ccShapeModelTemplate<ccCircle> ccCircleModel;
```

ccAnnulusModel

```
typedef ccShapeModelTemplate<ccAnnulus> ccAnnulusModel;
```

ccEllipse2Model
```
typedef ccShapeModelTemplate<ccEllipse2> ccEllipse2Model;
```

ccEllipseAnnulusModel
```
typedef ccShapeModelTemplate<ccEllipseAnnulus>
  ccEllipseAnnulusModel;
```

ccEllipseAnnulusSectionModel
```
typedef ccShapeModelTemplate<ccEllipseAnnulusSection>
  ccEllipseAnnulusSectionModel;
```

ccEllipseArc2Model
```
typedef ccShapeModelTemplate<ccEllipseArc2>
  ccEllipseArc2Model;
```

ccGenRectModel
```
typedef ccShapeModelTemplate<ccGenRect> ccGenRectModel;
```

ccGenAnnulusModel
```
typedef ccShapeModelTemplate<ccGenAnnulus>
  ccGenAnnulusModel;
```

ccPolylineModel
```
typedef ccShapeModelTemplate<ccPolyline> ccPolylineModel;
```

ccAffineRectangleModel
```
typedef ccShapeModelTemplate<ccAffineRectangle>
  ccAffineRectangleModel;
```

ccGenPolyModel
```
typedef ccShapeModelTemplate<ccGenPoly> ccGenPolyModel;
```

cc2WireframeModel
```
typedef ccShapeModelTemplate<cc2Wireframe>
  cc2WireframeModel;
```

ccBezierCurveModel
```
typedef ccShapeModelTemplate<ccBezierCurve>
  ccBezierCurveModel;
```

ccDeBoorSplineModel
```
typedef ccShapeModelTemplate<ccDeBoorSpline>
  ccDeBoorSplineModel;
```

ccShapeModelTemplate<>

ccInterpSplineModel

```
typedef ccShapeModelTemplate<ccInterpSpline>
 ccInterpSplineModel;
```

ccHermiteSplineModel

```
typedef ccShapeModelTemplate<ccHermiteSpline>
 ccHermiteSplineModel;
```

ccShapePerimData

```
#include <ch_cvl/shapedat.h>

template <class D> class ccShapePerimData:
                                        public virtual ccPersistent,
                                        public virtual ccRepBase
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This is the templated abstract base class for associating data with CVL shape descriptions. Association of data can be done for whole shape descriptions or for ranges of shape perimeters. This class is templated by the data type so that any kind of data can be associated with CVL shape descriptions. Classes derived from this base class should override the virtual member function **get()**.

The **ccShapePerimData** class can be instantiated by a class D that should have the following defined as public: default constructor, copy constructor, assignment operator, destructor, operator== for comparison, and operatorll for persistence.

Constructors/Destructors

ccShapePerimData

```
explicit ccShapePerimData(const D &defaultData = D());

virtual ~ccShapePerimData();
```

- `explicit ccShapePerimData(const D &defaultData = D());`

  Constructs a **ccShapePerimData** object with the given parameter.

- `virtual ~ccShapePerimData();`

  Destructor.

Operators

operator==

```
virtual bool operator== (const ccShapePerimData<D> &that)
                                                    const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

operator!=

```
virtual bool operator!= (const ccShapePerimData<D> &that)
                                                    const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

defaultData

```
void defaultData(const D & data);

D defaultData() const;
```

- `void defaultData(const D & data);`

  Sets the default data.

  Parameters

  data The new default data.

- `D defaultData() const;`

  Returns the default data. The default is **D()**.

get

```
virtual bool get(
  const ccPerimPos& perimPos, D &data) const;
```

Finds the associated shape data at the specified perimeter position. If found, the function assigns the found data to *data*, and returns true. If not found, it assigns the default data to *data* and returns false.

Parameters

perimPos The perimeter position. *perimPos* must be valid for this shape description.

data The location where the function places the data.

Throws

ccShapePerimDataDefs::BadParams
If *perimPos* is detectably invalid for the shape description.

Note: Derived classes that implement this function might throw.

clone

```
virtual ccPtrHandle<ccShapePerimData<D> > clone()
                                                    const = 0;
```

Returns a pointer to a new copy of this **ccShapePerimData** object.

Typedefs

ccShapeMask

```
typedef ccShapePerimData<ccShapeMaskValue> ccShapeMask;
```

ccShapeMaskPtrh

```
typedef ccPtrHandle<ccShapeMask> ccShapeMaskPtrh;
```

ccShapeMaskPtrh_const

```
typedef ccPtrHandle_const<ccShapeMask>
                                          ccShapeMaskPtrh_const;
```

ccShapeTol

```
typedef ccShapePerimData<ccBoundaryTol> ccShapeTol;
```

ccShapeTolPtrh

```
typedef ccPtrHandle<ccShapeTol> ccShapeTolPtrh;
```

ccShapeTolPtrh_const

```
typedef ccPtrHandle_const<ccShapeTol>
                                          ccShapeTolPtrh_const;
```

ccShapePerimData

ccShapePerimDataTable

```
#include <ch_cvl/shapedat.h>

template <class D> class ccShapePerimDataTable :
                                        public ccShapePerimData<D>
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This is a templated concrete class that associates the given shape perimeter ranges with the given data on the shape using a lookup table mechanism. The lookup table uses shape perimeter range vectors and data vectors to find the data value corresponding to a given shape perimeter position.

The shape perimeter ranges are represented by a start perimeter position and a signed range length. The shape perimeter range will not extend beyond the contour containing the start position. For example, perimeter ranges never jump gaps between contours. The perimeter length will be clipped if necessary to achieve this constraint. If clipping occurs, one endpoint of the perimeter range will correspond to the first or last point of the original contour that contains the perimeter range.

You provide a vector of data values. The data value and the perimeter range at index i of their respective vectors correspond to each other.

Note If you wish to associate a global data value with all perimeter ranges, leave the *perimRanges* and *rangeData* vectors empty and all perimeter ranges will use the *defaultData* value.

The **ccShapePerimDataTable** class can be instantiated by a class D that should have the following defined as public: default constructor, copy constructor, assignment operator, destructor, operator== for comparison, and operatorll for persistence.

Note that **ccShapePerimDataTable** becomes invalid for a shape description if the shape description is geometrically modified by clipping, a transformation, and so on.

Constructors/Destructors

ccShapePerimDataTable

```
explicit ccShapePerimDataTable(
  const ccShape &shape = cc2Point(),
  const D & defaultData = D(),
```

```
const cmStd vector<ccPerimRange> &perimRanges =
                                    cmStd vector<ccPerimRange>(0),
const cmStd vector<D> &rangeData = cmStd vector<D>(0));
```

Constructs a shape data lookup table object with the given parameters.

Parameters

shape The shape description containing the perimeter ranges.

defaultData The data returned when a perimeter range position is not found See **get()** below.

perimRanges The shape perimeter ranges that represent the shape boundary. Elements of *perimRanges* must be valid for the *shape description* and perimeter ranges must not overlap.

rangeData Data values corresponding to *perimRanges*.

Throws

ccShapePerimDataDefs::BadParams
If any perimeter range values are detectably invalid for *shape description*,
or if the vectors *perimRanges* and *rangeData* vectors are not the same size,
or if the *shape description* is infinite and not a primitive.

Operators

operator==

```
virtual bool operator== (
  const ccShapePerimData<D> &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

get

```
virtual bool get(
  const ccPerimPos& perimPos,
  D &data) const;
```

Returns the associated shape data at the perimeter position. This function finds the first perimeter range in the range vector that includes the given perimeter position. If such a range is found, the corresponding data from the data vector is assigned to *data* and this function returns true. If such a range is not found, then this function returns false and the default data is assigned to *data*.

Perimeter ranges must not overlap.

This function overrides the **ccShapePerimData::get()**.

Parameters

perimPos	A perimeter position. *perimPos* must be valid for the shape description.
data	Where to put the data.

Throws

ccShapePerimDataDefs::BadParams
If *perimPos* is detectably invalid for the shape description.

setShapeAndVectors

```
void setShapeAndVectors(
  const ccShape &shape,
  const cmStd vector<ccPerimRange> &perimRanges,
  const cmStd vector<D> &rangeData);
```

Sets the shape description, shape perimeter range, and range data vectors.

Elements of *perimRanges* must be valid for the *shape description*.

Parameters

shape	The shape description containing the perimeter ranges.
perimRanges	A vector of perimeter ranges.
rangeData	A vector of range data corresponding to *perimRanges*.

Throws

ccShapePerimDataDefs::BadParams

If any range values are detectably invalid for *shape description,* or if *perimRanges* and *rangeData* vector sizes are not the same, or if the *shape description* is infinite and not a primitive.

Note: The state of this object remains unchanged if it throws.

shape

```
ccShapePtrh_const shape() const;
```

Returns the shape description of this object.

perimRanges

```
const cmStd vector<ccPerimRange> &perimRanges() const;
```

Returns the perimeter range vector of this object.

rangeData

```
const cmStd vector<D> &rangeData() const;
```

Returns the range data vector of this object.

Macros

cmShapePerimDataClone

```
cmShapePerimDataClone(
  ccShapePerimDataTable<D>, ccShapePerimData<D>);
```

Overrides **ccShapePerimData::clone()**.

Typedefs

ccShapeMaskTable

```
typedef ccShapePerimDataTable<ccShapeMaskValue>
                                              ccShapeMaskTable;
```

ccShapeTolTable

```
typedef ccShapePerimDataTable<ccBoundaryTol>
                                                ccShapeTolTable;
```

ccShapeTolStats

```
#include <ch_cvl/shapetol.h>

class ccShapeTolStats: public ccShapeTol
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccShapeTolStats is a class for estimating local shape boundary tolerances by training on images that contain non-defective boundaries. **ccShapeTolStats** is used with the Boundary Inspector tool.

When you train the Boundary Inspector tool you must provide a **ccShapeTol** object that specifies the allowed tolerances. Cognex provides this **ccShapeTolStats** tool for obtaining these tolerances automatically from good training images through statistical calculations. You can also provide your own tolerances if you wish to obtain them another way.

Constructors/Destructors

ccShapeTolStats

```
ccShapeTolStats();

ccShapeTolStats(
  const ccShape &modelBoundary,
  const cc2XformBase &clientFromImage,
  const ccBoundaryTol &defaultTol =
     ccBoundaryTol(ccRange::EmptyRange(), ccRadian(0.0)),
  const ccShapeTolStatsParams &statsParams =
                                    ccShapeTolStatsParams(),
  const ccShapeTolStatsModelParams &modelParams =
                                ccShapeTolStatsModelParams(),
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);

ccShapeTolStats(
  const ccShape &modelBoundary,
  const ccShapeMask &shapeMask,
  const cc2XformBase &clientFromImage,
  const ccBoundaryTol &defaultTol =
       ccBoundaryTol(ccRange::EmptyRange(), ccRadian(0.0)),
```

```
  const ccShapeTolStatsParams &statsParams =
                                        ccShapeTolStatsParams(),
  const ccShapeTolStatsModelParams &modelParams =
                                    ccShapeTolStatsModelParams(),
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);

ccShapeTolStats(const ccShapeTolStats &that);

~ccShapeTolStats();
```

Constructs a **ccShapeTolStats** object using the given parameters.

Boundaries or boundary portions without a defined tangent angle are ignored by this tool.

A valid clipping region is a closed, convex **ccPolyline** with at least 3 vertices.

- `ccShapeTolStats();`

  Constructs a default **ccShapeTolStats** object.

- ```
  ccShapeTolStats(
    const ccShape &modelBoundary,
    const cc2XformBase &clientFromImage,
    const ccBoundaryTol &defaultTol =
       ccBoundaryTol(ccRange::EmptyRange(), ccRadian(0.0)),
    const ccShapeTolStatsParams &statsParams =
                                        ccShapeTolStatsParams(),
    const ccShapeTolStatsModelParams &modelParams =
                                    ccShapeTolStatsModelParams(),
    ccDiagObject* obj = 0,
    c_UInt32 diagFlags = 0);
  ```

  Constructs a **ccShapeTolStats** object from the parameters provided. Does not use a shape mask.

  Parameters

modelBoundary	The shape description containing the model boundary.
clientFromImage	The transform from image space to client space.
defaultTol	Use this default if there is not enough information to compute a statistical tolerance.
statsParams	The statistical training parameters.
modelParams	The statistical model parameters.
  ```

obj A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccShapePerimDataDefs::BadParams
If the shape description is infinite and not a primitive, or if the shape description is infinite and *clipRegion* is not a closed polyline with at least 3 vertices.

-
```
ccShapeTolStats(
  const ccShape &modelBoundary,
  const ccShapeMask &shapeMask,
  const cc2XformBase &clientFromImage,
  const ccBoundaryTol &defaultTol =
      ccBoundaryTol(ccRange::EmptyRange(), ccRadian(0.0)),
  const ccShapeTolStatsParams &statsParams =
                                      ccShapeTolStatsParams(),
  const ccShapeTolStatsModelParams &modelParams =
                                  ccShapeTolStatsModelParams(),
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);
```

Constructs a **ccShapeTolStats** object from the parameters provided, including a shape mask.

Parameters

modelBoundary The shape description containing the model boundary.

shapeMask A mask applied to the shape boundary that selects only the boundary parts you wish to train.

clientFromImage The transform from image space to client space.

defaultTol Use this default if there is not enough information to compute a statistical tolerance.

statsParams The statistical training parameters.

modelParams The statistical model parameters.

obj A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccShapePerimDataDefs::BadParams
If the shape description is infinite and not a primitive,
or if the shape description is infinite and *clipRegion* is not a closed polyline with at least 3 vertices,
or if *shapeMask* is detectably invalid for the model boundary.

- `ccShapeTolStats(const ccShapeTolStats &that);`

  Copy constructor; performs a deep copy.

- `~ccShapeTolStats();`

  Destructor.

Operators

operator== `virtual bool operator==(const ccShapeTol &that) const;`

Equality operator. Returns true if the internal data members of this object are exactly equal to the internal data members of *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

operator==

```
virtual bool operator=(const ccShapeTol &that) const;
```

Deep copy assignment operator. Make this object a copy of *that*.

Public Member Functions

modelParams

```
ccShapeTolStatsModelParams modelParams() const;

void modelParams(
  const ccShapeTolStatsModelParams &modelParams);
```

The statistical model parameters determine how the data collected by training will be interpreted to estimate boundary tolerances.

The default *modelParams* is **ccShapeTolStatsModelParams()**.

- ```
  ccShapeTolStatsModelParams modelParams() const;
  ```

  Returns the current statistical model parameters.

- ```
  void modelParams(
    const ccShapeTolStatsModelParams &modelParams);
  ```

  Sets new statistical model parameters.

  Parameters

  modelParams The new parameters.

statsParams

```
ccShapeTolStatsParams statsParams() const;
```

Returns the statistical tolerance estimation parameters which are used for clipping the model boundary and determining the capture range between the model boundary and the image contours. The default is **ccShapeTolStatsParams()**.

ccShapeTolStats

defaultTol

```
ccBoundaryTol defaultTol() const;

void defaultTol(const ccBoundaryTol &defaultTol);
```

The default tolerance is returned for perimeter positions for which the statistical training process does not have enough information for a tolerance estimate.

The default is **ccBoundaryTol(ccRange::EmptyRange(), ccRadian(0.0))**.

- `ccBoundaryTol defaultTol() const;`

  Returns the current default boundary tolerance.

- `void defaultTol(const ccBoundaryTol &defaultTol);`

  Sets a new default boundary tolerance.

  Parameters

  defaultTol The new default boundary tolerance.

addImage

```
void addImage(
  const ccPelBuffer_const<c_UInt8> &image,
  const cc2XformBase &pose,
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);

void addImage(
  const ccPelBuffer_const<c_UInt8> &image,
  const cc2XformBase &pose,
  const ccFeatureletFilter &featureletFilter,
  ccDiagObject* obj = 0,
  c_UInt32 diagFlags = 0);
```

Finds the image contours in the given image, filters the found contours with *featureletFilter* (if one is provided), and matches the remaining contours with the model boundary transformed by *pose*. The results from the matching operation are used to update the statistics about the local boundary tolerances.

The scaling factors of the transform pose should be close to 1.0 for best tool performance.

- ```
  void addImage(
    const ccPelBuffer_const<c_UInt8> &image,
    const cc2XformBase &pose,
    ccDiagObject* obj = 0,
    c_UInt32 diagFlags = 0);
  ```

  Adds an image without featurelet filtering.

  Parameters

image	The image to add.
pose	The model space to client space transform.
obj	A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.
diagFlags	Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

  Throws

  ccShapePerimDataDefs::BadParams
  If *image* is not bound.

  Note: The state of this object remains unchanged if it throws.

- ```
  void addImage(
    const ccPelBuffer_const<c_UInt8> &image,
    const cc2XformBase &pose,
    const ccFeatureletFilter &featureletFilter,
    ccDiagObject* obj = 0,
    c_UInt32 diagFlags = 0);
  ```

  Adds an image using featurelet filtering.

  The *featureletFilter* is applied to the image contours after the contours are transformed into the coordinate system in which the model boundary is defined.

Parameters

image	The image to add.
pose	The model space to client space transform.
featureletFilter	The featurelet filter to be applied to the image.
obj	A container class object that holds diagnostic records created during the inspection. *obj* must point to a valid memory location or it must be NULL. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.
diagFlags	Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

Throws

ccShapePerimDataDefs::BadParams
If *image* is not bound.

Note: The state of this object remains unchanged if it throws.

get

```
virtual bool get(
  const ccPerimPos& perimPos,
  ccBoundaryTol &tol) const;
```

Returns the associated shape tolerance estimated at the perimeter position. The tolerance estimation depends on the statistical model parameters. If there is enough statistical data collected for the given perimeter position, this function returns true and the estimated boundary tolerance is assigned to *tol*. Otherwise, this function returns false and default tolerance is assigned to *tol*.

This function overrides the **ccShapeTol::get()**.

Parameters

perimPos	The perimeter position where the shape tolerance is obtained.
tol	Where the shape tolerance is returned.

Throws

ccShapePerimDataDefs::BadParams

If *perimPos* is detectably invalid for **shape()**.

modelBoundary

```
ccShapePtrh_const modelBoundary() const;
```

Returns the model boundary. The default model boundary is **cc2Point()**.

clientFromImage

```
cc2XformBasePtrh_const clientFromImage() const;
```

Returns the client-from-image transform. The relationship between the client and image space of the boundary model is specified by this transform. The default is the identity transform.

shapeMask

```
ccShapeMaskPtrh_const shapeMask() const;
```

Returns the shape mask. The portions of the model boundary masked by this shape mask are ignored by this tool.

The default **ccShapeTolStats** object does not include a shape mask.

Throws

ccShapePerimDataDefs::NotAvailable

If a shape mask was not provided when this object was created.

Macros

cmShapePerimDataClone

```
cmShapePerimDataClone(ccShapeTolStats, ccShapeTol);
```

Overrides the **ccShapePerimData::clone()**.

ccShapeTolStats

ccShapeTolStatsModelParams

```
#include <ch_cvl/shapetol.h>

class ccShapeTolStatsModelParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccShapeTolStatsModelParams is a container class that includes parameters to be used in interpreting the statistical data collected by the **ccShapeTolStats** class. You pass in a **ccShapeTolStatsModelParams** object when you instantiate a **ccShapeTolStats** object.

There are two modes for interpreting the statistical data:

Normal Distribution Model assumes the data collected by **ccShapeTolStats** can be modeled by normal distribution. This mode requires that you provide a valid confidence level so that the boundary tolerances can be calculated from the collected data.

Non-parametric Model assumes that the data collected by **ccShapeTolStats** cannot be modeled by any parametric distribution. Only the observed maximal deviations from the model boundary and the *slack tolerances* you provide are used to estimate the boundary tolerances.

Constructors/Destructors

ccShapeTolStatsModelParams

```
explicit ccShapeTolStatsModelParams(
  StatisticalModel statsM =
                ccShapeTolStatsModelParams::kDefaultModel);
```

Constructs a **ccShapeTolStatsModelParams** object with the given parameter.

Parameters

statsM The statistical model. Must be one of the **StatisticalModel** enums.

Enumerations

StatisticalModel `enum StatisticalModel`

Defines the modes used in interpreting the statistical data.

Value	Meaning
eNormalDistribution = 0	The data can be modeled by normal distribution
eNonParametric = 1	The data cannot be modeled by any parametric distribution
kDefaultModel = *eNormalDistribution*	The default mode

Operators

operator==

```
bool operator== (
  const ccShapeTolStatsModelParams &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

operator!=

```
bool operator!= (
  const ccShapeTolStatsModelParams &that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

statsModel

```
ccShapeTolStatsModelParams::StatisticalModel statsModel()
                                                    const;
```

Returns the statistical model specified at construction time.

confidenceLevel

```
double confidenceLevel() const;
void confidenceLevel(double confidenceLevel);
```

The confidence level is used as the statistical confidence level in the statistical decision process of estimating boundary tolerances. The confidence level is a number in the range 0.5 through 1.0.

The default confidence level value is 0.95.

Notes

This function is used in Normal Distribution mode only.

- `double confidenceLevel() const;`

  Returns the current confidence level.

- `void confidenceLevel(double confidenceLevel);`

  Sets a new confidence level.

  Parameters

  confidenceLevel The new confidence level.

  Throws

  ccShapePerimDataDefs::BadParams

  If *confidenceLevel* < 0.5 or > 1.0.

  Note: The state of this object remains unchanged if it throws.

slackTol

```
ccBoundaryTol slackTol() const;
void slackTol(const ccBoundaryTol &slackTol);
```

A small additional tolerance added to the observed maximum tolerance to establish the tolerance range used during inspections. This extends the range slightly so that inspection computations are not made on the range boundaries.

The default slack tolerance is **ccBoundaryTol(ccRange::EmptyRange, ccRadian(0.0))**.

Notes

This function is used in Non-parametric Model mode only.

- `ccBoundaryTol slackTol() const;`

  Returns the slack tolerance.

- `void slackTol(const ccBoundaryTol &slackTol);`

  Sets the slack tolerance.

  Parameters

  slackTol The new slack tolerance.

ccShapeTolStatsParams

```
#include <ch_cvl/shapetol.h>

class ccShapeTolStatsParams
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This is a container class that includes parameters to be used by the **ccShapeTolStats** class for statistical training of the Boundary Inspector tool. There are two parameters: *captureRange* and *clipRegion*.

The capture range specifies a tolerance range where you expect to find contours in training images. All contours outside this range are ignored when computing statistical tolerances. The clip region allows you to specify where on the model statistical tolerances are required.

Constructors/Destructors

ccShapeTolStatsParams

```
explicit ccShapeTolStatsParams(
  const ccBoundaryTol &captureRange =
        ccBoundaryTol(ccRange(-1., 1.), ccRadian(ckPI_4)),
  const ccPolyline &clipRegion = ccPolyline(true));
```

Constructs a **ccShapeTolStatsParams** object with the given parameters.

To enable clipping *clipRegion* must be a closed, convex polyline. Otherwise no clipping is performed.

Parameters

captureRange The capture range. Image contours found outside the capture range are ignored.

clipRegion The clipping region. A closed convex polyline that encloses all portions of the model where statistical boundary tolerances are required.

Throws

ccShapePerimDataDefs::BadParams
If the clipping region is open.

Operators

operator==

```
bool operator== (const ccShapeTolStatsParams &that) const;
```

Equality test operator. Returns true if this object is exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

operator!=

```
bool operator!= (const ccShapeTolStatsParams &that) const;
```

Inequality test operator. Returns true if this object is not exactly equal to *that*. Returns false otherwise.

Parameters

that The object to compare with this object.

Public Member Functions

clipRegion

```
const ccPolyline& clipRegion() const;

void clipRegion(const ccPolyline &clipRegion);
```

Defines the clipping region. To perform clipping, the clipping region must be a closed convex polyline. When clipping is performed, the model boundary is clipped by this region and boundary tolerances are estimated only for this clipped portion of the model.

No clipping is performed unless a valid clipping region is provided here. A valid clipping region is a closed, convex **ccPolyline** with at least 3 vertices.

The default clipping region is a closed polyline with no vertices (no clipping).

The clipping region is defined in the model coordinates of the model boundary.

- `const ccPolyline& clipRegion() const;`

  Returns the current clipping region.

- `void clipRegion(const ccPolyline &clipRegion);`

  Sets a new clipping region.

  Parameters

  clipRegion The new clipping region.

Throws

ccShapePerimDataDefs::BadParams
If the clipping region is open.

captureRange

```
ccBoundaryTol captureRange() const;
void captureRange(const ccBoundaryTol &captureRange);
```

No valid image contours from training images are expected to be outside the capture range you specify. If there are such contours, they will be ignored in the statistical estimation of boundary tolerances.

The default capture range is **ccBoundaryTol(ccRange(-1.0, 1.0), ccRadian(ckPI_4))**.

- `ccBoundaryTol captureRange() const;`

  Returns the current capture range.

- `void captureRange(const ccBoundaryTol &captureRange);`

  Sets a new capture range.

  Parameters

  captureRange The new capture range.

ccShapeTolStatsParams

ccShapeTree

```
#include <ch_cvl/shaptree.h>

class ccShapeTree : public ccShape;
```

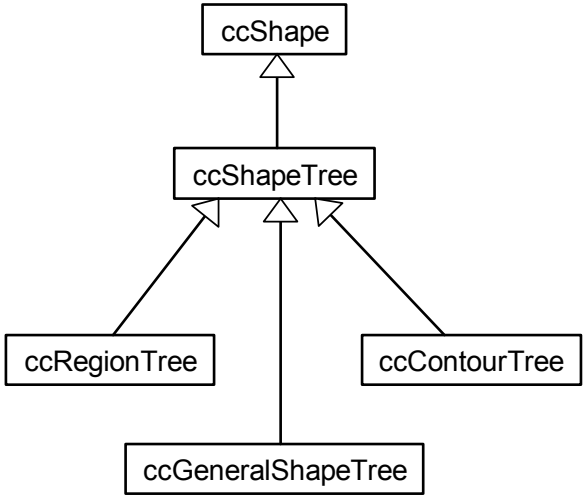
Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

The **ccShapeTree** class is the abstract base class for all hierarchical shapes. A **ccShapeTree** may have any number of **ccShape** children. Any of these children can themselves be **ccShapeTree**s, as **ccShapeTree** derives from **ccShape**.

Notes

A *primitive* shape is a **ccShape** that does not derive from **ccShapeTree**, such as a **ccLineSeg** or **ccCircle**.



The above figure shows the **ccShapeTree** class inheritance hierarchy.

Shared Children, Copying and Transforming Trees

Shape trees and classes derived from them are implemented with internal pointer handles to children. These pointer handles are **const** (that is, **ccShapePtrh_const**), so it is impossible to modify children objects through their parent, even if the parent is not **const**. It follows then that no object in a tree can be modified once it is placed into the tree.

Copying and cloning always use shallow copying of children. Thus, the original and the copy point to the same set of children objects; the copy is achieved by copying pointer handles. Trees cannot be modified in place. Instead, a tree modification is accomplished by creating a new tree using parts of the old one, possibly re-arranged and modified. Every modification of a tree node generates a new tree node. Nodes that don't change are simply copied or cloned. As they are shallow, copying and cloning are fast operations. This is the most efficient approach.

As an example, consider the following algorithm for mapping an entire tree by an affine transform. This function takes an original tree *T* and an affine transform *X*, and returns a new tree *T'* that is the transform of the original.

1. Recursively map the children of *T*, obtaining a set of new, transformed children.
2. Set *T'* to be a clone of *T*. This is a fast operation as it is involves a shallow copy. After this operation, *T'* and *T* share the same children.
3. Replace the children of *T'* with the set of transformed children computed in step 1. The children of *T* remain unchanged.
4. Return *T'*.

Copy Insertion versus Direct Insertion

Insertions of children into trees are typically done by *copy insertion*, that is, a copy of the object to be inserted is generated using **clone()**, and this copy is inserted into the tree. Since **clone()** is shallow, this is still a fast operation.

The alternative is *direct insertion*. You can insert a pointer handle directly into the list of children of a tree without copying anything but a single pointer handle. However, it is critical that objects added by direct insertion are never subsequently modified. Doing so violates the assumptions of the shape tree framework and can cause undefined behavior. For this reason, and because the efficiency gains are typically slight, Cognex strongly discourages the use of direct insertion.

There are two situations in which direct insertion may be substantially faster than copy insertion. The first is when the child to be inserted is a **ccShapeTree** with N children, where N is very large, for example a thousand or more. Direct insertion requires one pointer handle copy; copy insertion requires N pointer handle copies. The second case is when the child to be inserted is a very large primitive shape, such as a **ccPolyline** with thousands of vertices. Copy insertion copies this large object; direct insertion does not. If either of these cases exist in an inner loop, direct insertion might be worthwhile.

Finally, direct insertion is useful in the implementation of optimized classes derived from **ccShapeTree**, where correct usage can be assured.

The same distinction applies to replacing children in trees. The new child is either copied into the tree or directly placed into it by a single pointer handle assignment. The trade-offs and caveats are the same as for insertion.

Constructors/Destructors

ccShapeTree `ccShapeTree();`

Default constructor. Constructs a **ccShapeTree** containing no children.

Notes

Copies are shallow, that is, they are achieved by copying pointer handles. Hence, copy construction leads to sharing of children between different **ccShapeTree**s.

Only derived classes call this constructor, as **ccShapeTree** is an abstract base class and is never instantiated directly.

Operators

operator= `ccShapeTree& operator=(const ccShapeTree &rhs);`

Assign *rhs* to this **ccShapeTree**.

Parameters

rhs	The **ccShapeTree** to assign to this one.

Notes

Assignments are shallow, that is, they are achieved by copying pointer handles. Hence, assignment leads to sharing of children between different **ccShapeTree**s.

This method is protected to disallow assignment of shapes through references to the base class.

operator== `bool operator==(const ccShapeTree &rhs) const;`

Returns true if and only if this **ccShapeTree** is equal to *rhs*. Two **ccShapeTree**s are equal if their hierarchical structures are the same, and all of the corresponding primitive shapes are equal, under those primitives shapes' own **operator==()**.

Parameters

rhs	The other **ccShapeTree**.

operator!= `bool operator!=(const ccShapeTree &rhs) const;`

Returns true if and only if this **ccShapeTree** is not equal to *rhs*.

Parameters

rhs	The other **ccShapeTree**.

Public Member Functions

numChildren `c_Int32 numChildren() const;`

Returns the number of children of this **ccShapeTree**.

isLeaf `bool isLeaf() const;`

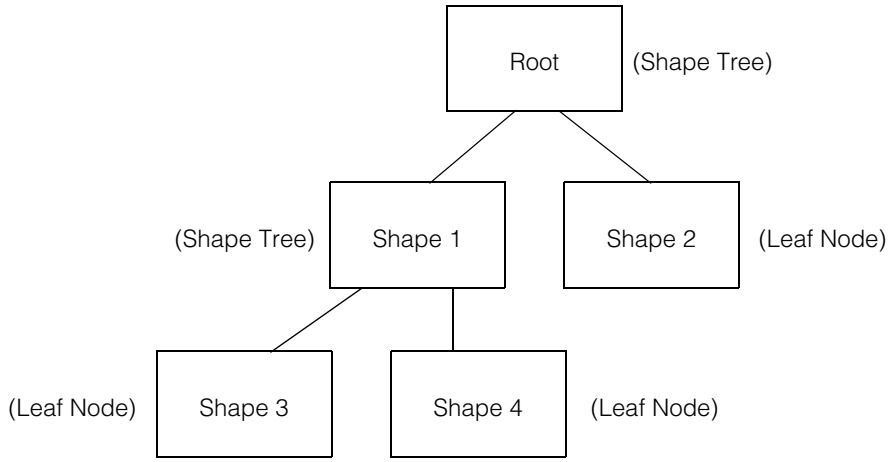
Returns true if and only if this **ccShapeTree** is a leaf node, that is, has no children.

size `c_Int32 size(bool primitivesOnly = false) const;`

Returns the number of descendants of this **ccShapeTree**, including the root node.

Parameters

primitivesOnly	If true, only primitive shapes are counted; otherwise all shapes in the hierarchy are counted. When *primitivesOnly* is true, only primitives that are leaf nodes are counted; primitives that are boundaries of **ccRegionTree**s are not counted. Therefore, this function always returns zero when invoked on a ccRegionTree with *primitivesOnly* set to true. This is the case because region trees have no primitive leaves.



The **ccShapeTree** structure shown above has four children. The **size()**, or number of nodes including the root, of this shape tree is five when the *primitivesOnly* flag is false; when *primitivesOnly* is true, the size is three. The **height()**, or longest path from the root to a leaf, of this shape tree is two.

height

```
c_Int32 height() const;
```

Returns the length of the longest path from the root of this **ccShapeTree** to a leaf. Leaf nodes have a height of zero.

child

```
const ccShape &child(c_Int32 idx) const;
```

Returns the indexed child of this **ccShapeTree**.

Parameters

idx The index.

Throws

ccShapesError::BadIndex

idx is less than zero or greater than or equal to **numChildren()**.

children

```
const cmStd vector<ccShapePtrh_const> &children() const;
```

Returns the children of this **ccShapeTree**. The returned object is a vector of pointer handles to the **ccShape** children.

insertChild

```
virtual void insertChild(c_Int32 idx,
  const ccShape &child);

virtual void insertChild(c_Int32 idx,
  const ccShapePtrh_const &child, bool direct = false);
```

- ```
  virtual void insertChild(c_Int32 idx,
    const ccShape &child);
  ```

  Inserts the **ccShape** referenced or pointed to by *child* as a new child of this **ccShapeTree**. The shape is inserted into the list of children so that after the insertion, it has index *idx*. Use an *idx* of 0 to insert the child at the front of the list, and use **numChildren()** to append it to the end of the list.

  Parameters

  idx — The index.

  child — The child to insert.

  Throws

  ccShapesError::BadIndex
  idx is less than zero or greater than **numChildren()** before insertion.

  ccShapesError::BadGeom
  Inserting the child would produce an invalid configuration of children. The default method never throws this exception.

- ```
  virtual void insertChild(c_Int32 idx,
    const ccShapePtrh_const &child, bool direct = false);
  ```

  Inserts the **ccShape** referenced or pointed to by *child* as a new child of this **ccShapeTree**. The shape is inserted into the list of children so that after the insertion, it has index *idx*. Use an *idx* of 0 to insert the child at the front of the list, and use **numChildren()** to append it to the end of the list.

  Parameters

  idx — The index.

  child — The child to insert.

  direct — If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it. See *Copy Insertion versus Direct Insertion* on page 2976 for details.

Notes

Inserting a child will change the indices of all existing children whose indices prior to the insertion are greater than or equal to *idx*.

A derived class may override this method to place constraints on the types of children it may have. An overload of this method may throw *ccShapesError::BadGeom* if the insertion would result in an invalid configuration of children.

The shape tree framework assumes that there are no cycles in the hierarchy, although it doesn't explicitly check this. Undefined behavior will result if *child* is an ancestor of this **ccShapeTree**.

Throws

ccShapesError::BadIndex
: *idx* is less than zero or greater than **numChildren()** before insertion.

ccShapesError::BadGeom
: Inserting the child would produce an invalid configuration of children. The default method never throws this exception.

insertChildren

```
virtual void insertChildren(c_Int32 idx,
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Inserts a set of **ccShape**s as children of this **ccShapeTree**. The shapes are inserted into the list of children contiguously so that after the insertion, they have indices *idx* through *idx* + **children.size()** - 1.

Parameters

idx	The index. Use an *idx* of zero to insert at the front of the list, use **numChildren()** to append at the end of the list.
children	The vector of children to insert.
direct	If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it. See *Copy Insertion versus Direct Insertion* on page 2976 for details.

Notes

Inserting children will change the indices of all existing children whose indices prior to the insertion are greater than or equal to idx.

A derived class may override this method to place constraints on the types of children it may have. An overload of this method may throw *ccShapesError::BadGeom* if the insertion would result in an invalid configuration of children. If such an exception is thrown, none of the children are inserted, even if some of them would be valid insertions.

The shape tree framework assumes that there are no cycles in the hierarchy, although it doesn't explicitly check this. Undefined behavior will result if any of the individual *children* is an ancestor of this **ccShapeTree**.

Throws

ccShapesError::BadIndex
: *idx* is less than zero or greater than **numChildren()** before insertion.

ccShapesError::BadGeom
: Inserting *children* would produce an invalid configuration of children. The default method never throws this exception.

---

addChild

```
virtual void addChild(const ccShape &child);

virtual void addChild(const ccShapePtrh_const &child,
  bool direct = false);
```

- `virtual void addChild(const ccShape &child);`

Convenience function for appending a shape to the end of this **ccShapeTree**'s list of children. Functionally equivalent to invoking **insertChild()** with *idx* equal to **numChildren()**.

Parameters

child The child to add.

Throws

ccShapesError::BadGeom
: Inserting the child would produce an invalid configuration of children. The default method never throws this exception.

- ```
  virtual void addChild(const ccShapePtrh_const &child,
    bool direct = false);
  ```

  Convenience function for appending a shape to the end of this **ccShapeTree**'s list of children. Functionally equivalent to invoking **insertChild()** with *idx* equal to **numChildren()**.

  Parameters

  child — The child to add.

  direct — If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it. See *Copy Insertion versus Direct Insertion* on page 2976 for details.

  Throws

  ccShapesError::BadGeom

  Inserting the child would produce an invalid configuration of children. The default method never throws this exception.

addChildren

```
virtual void addChildren(
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Convenience function for inserting a set of shapes at the end of this **ccShapeTree**'s list of children. Functionally equivalent to invoking **insertChildren()** with *idx* equal to **numChildren()**.

Parameters

children — The children to add.

direct — If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it. See *Copy Insertion versus Direct Insertion* on page 2976 for details.

Throws

ccShapesError::BadGeom

Inserting *children* would produce an invalid configuration of children. The default method never throws this exception.
```

replaceChild

```
virtual void replaceChild(c_Int32 idx,
 const ccShape &child);

virtual void replaceChild(c_Int32 idx,
 const ccShapePtrh_const &child, bool direct = false);
```

- ```
  virtual void replaceChild(c_Int32 idx,
   const ccShape &child);
  ```

  Replaces the indexed child of this **ccShapeTree** with the given shape.

  Parameters

  idx The index.

  child The child replacing the indexed child.

- ```
  virtual void replaceChild(c_Int32 idx,
   const ccShapePtrh_const &child, bool direct = false);
  ```

  Replaces the indexed child of this **ccShapeTree** with the given shape.

  Parameters

  idx The index.

  child The child replacing the indexed child.

  direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it. See *Copy Insertion versus Direct Insertion* on page 2976 for details.

  Notes

  A derived class may override this method to place constraints on the types of children it may have. An overload of this method may throw *ccShapesError::BadGeom* if the replacement would result in an invalid configuration of children.

  The shape tree framework assumes that there are no cycles in the hierarchy, although it doesn't explicitly check this. Undefined behavior will result if *child* is an ancestor of this **ccShapeTree**.

  Throws

  ccShapesError::BadIndex

  idx is less than zero or greater than or equal to **numChildren()**.

  ccShapesError::BadGeom

  Inserting the child would produce an invalid configuration of children. The default method never throws this exception.

replaceChildren

```
virtual void replaceChildren(
  const cmStd vector<ccShapePtrh_const> &children,
  bool direct = false);
```

Replace the children of this **ccShapeTree** with the given set of children.

Parameters

children The vector of children replacing the current children.

direct If false (the default), copy insertion is performed; if true, direct insertion is performed. Cognex recommends always using copy insertion unless there is a compelling reason not to use it. See *Copy Insertion versus Direct Insertion* on page 2976 for details.

Notes

A derived class may override this method to place constraints on the types of children it may have. An overload of this method may throw *ccShapesError::BadGeom* if the replacement would result in an invalid configuration of children. In this case, this **ccShapeTree** is left unchanged.

The shape tree framework assumes that there are no cycles in the hierarchy, although it doesn't explicitly check this. Undefined behavior will result if any of the individual children is an ancestor of this **ccShapeTree**.

Throws

ccShapesError::BadGeom

Replacing the children would produce an invalid configuration of children. The default method never throws this exception.

removeChild

```
virtual void removeChild(c_Int32 idx);
```

Removes the indexed child of this **ccShapeTree**.

Parameters

idx The index.

Notes

Removing a child will change the indices of all remaining children whose indices prior to the removal are greater than *idx*.

A derived class may override this method to place constraints on the types of children it may have. An overload of this method may throw ccShapesError::BadGeom if the removal would result in an invalid configuration of children.

Throws

ccShapesError::BadIndex

idx is less than zero or greater than or equal to **numChildren()** before removal.

ccShapesError::BadGeom

Removing the child would produce an invalid configuration of children. The default method never throws this exception.

removeChildren

```
virtual void removeChildren();
```

Removes all children of this **ccShapeTree**.

Notes

A derived class may override this method to place constraints on the types of children it may have. An overload of this method may throw *ccShapesError::BadGeom* if the derived type must have at least one child.

Throws

ccShapesError::BadGeom

Removing the children would produce an invalid configuration of children. The default method never throws this exception.

flatten

```
ccShapeTreePtrh flatten(bool preserveContours = false)
  const;
```

Computes and returns a flattened version of this shape tree. The flattened tree contains all the primitive shapes of the original tree, including the primitives defining the boundaries of region trees. These primitives are all arranged as direct children of the root node, which is the only shape tree node in the flattened tree unless the *preserveContours* argument is true.

If this tree is a contour tree, then the returned tree is also a contour tree. Otherwise, the returned tree is a general shape tree.

Parameters

preserveContours

Whether contours should be preserved when flattening this shape tree.

Notes

If *preserveContours* is true, the connectivity of shapes within contour tree descendants of this shape is preserved when the tree is flattened. That is, if C is a contour tree that is a proper descendant of this shape tree and C's parent is not a contour tree, then the returned tree will contain a flattened version of C as a child. The same would apply if C were the boundary of a region tree. In either case, the returned tree will not be completely flat, as it may contain contour tree children.

However, such children will have only primitives as their own children. In other words, the returned tree can have a height of at most two. The preserve contours mode is useful when a tree should be flattened to the fullest extent possible without losing connectivity information along the contours.

The order of the primitives in the flattened tree is the order in which they are encountered in a standard depth-first, pre-order traversal of this tree. The distinction between pre-, and post-order traversal is significant only when there are primitives as internal nodes, as is the case, for example, when region trees are present.

isEmpty `virtual bool isEmpty() const;`

Returns true if and only if all primitive shapes in the hierarchy are empty.

Notes

A **ccContourTree** or **ccGeneralShapeTree** is empty if and only if it contains no children, or all of its children are empty. A **ccRegionTree** can never be empty, because it must contain a boundary that is a region and region shapes are never empty.

See **ccShape::isEmpty()** for more information.

hasTangent `virtual bool hasTangent() const;`

Returns true if and only if any of the primitive shapes in the hierarchy has a tangent. See **ccShape::hasTangent()** for more information.

isDecomposed `virtual bool isDecomposed() const;`

Returns true if all primitive shapes in the hierarchy are already decomposed. See **ccShape::isDecomposed()** for more information.

boundingBox `virtual ccRect boundingBox() const;`

Returns the enclosing rectangle of this **ccShapeTree**.

Notes

The enclosing rectangle of this **ccShapeTree** is the smallest rectangle that encloses all of its component primitives.

See **ccShape::boundingBox()** for more information.

perimeter

```
virtual double perimeter() const;
```

Returns the perimeter of this shape tree.

See **ccShape::perimeter()** for more information.

nearestPerimPos

```
virtual ccPerimPos nearestPerimPos(const ccShapeInfo& info,
  const cc2Vect& point) const;
```

Returns the nearest perimeter position on this shape tree to the given point, as determined by **nearestPoint()**.

Parameters

info	Shape information for this shape tree.
point	The point.

See **ccShape::nearestPerimPos()** for more information.

mapShape

```
virtual ccShapePtrh mapShape(const cc2Xform& X) const;
```

Returns this ccShapeTree mapped by X.

Parameters

X	The transformation object.

Notes

Generates a new **ccShapeTree** that is an hierarchical refinement of the original. That is, the original hierarchy is replicated up to the leaf nodes. All primitive shapes in the original are mapped by *X*, and this mapping may convert the primitives themselves to subhierarchies.

decompose

```
virtual ccShapePtrh decompose() const;
```

Returns a new shape tree describing the same geometry as this **ccShapeTree**, but using only a subset of shape primitives. See **ccShapeTree::decompose()** for more information.

ccSharpnessParams

```
#include <ch_cvl/imgsharp.h>

class ccSharpnessParams
```

This class encapsulates parameters passed to the **cfImageSharpness()** global function that specify how the function determines a sharpness score for an image.

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

Constructors/Destructors

ccSharpnessParams

```
ccSharpnessParams();
```

Default constructor for sharpness parameters. The following defaults are set:

Parameter	Default value
algorithm	*eAutoCorrelation*
sx	1
sy	1
freqBand	0.25 through 0.45
noiseLevel	0.0
lowPassSmoothing	0

Enumerations

Algorithm `enum Algorithm`

This enumeration defines the sharpness algorithms.

Value	Meaning	Parameter used
eAutoCorrelation = 1	Autocorrelation algorithm	*noiseLevel*
eAbsDiff	Absolute difference algorithm	*sx, sy*
eEdgeGradient	Edge gradient algorithm **Note**: This algorithm is deprecated	
eBandPass	Frequency band response algorithm	*freqBand*
eGradientEnergy[1]	Gradient energy	*lowPassSmoothing*

(1) Most applications achieve the best results using this algorithm.

Public Member Functions

algorithm

```
ccSharpnessParams::Algorithm algorithm() const;
void algorithm(ccSharpnessParams::Algorithm algorithm);
```

Specifies the algorithm to be used by the **cfImageSharpness()** global function.

- `ccSharpnessParams::Algorithm algorithm() const;`

  Returns the current algorithm parameter.

- `void algorithm(ccSharpnessParams::Algorithm algorithm);`

  Sets a new algorithm parameter.

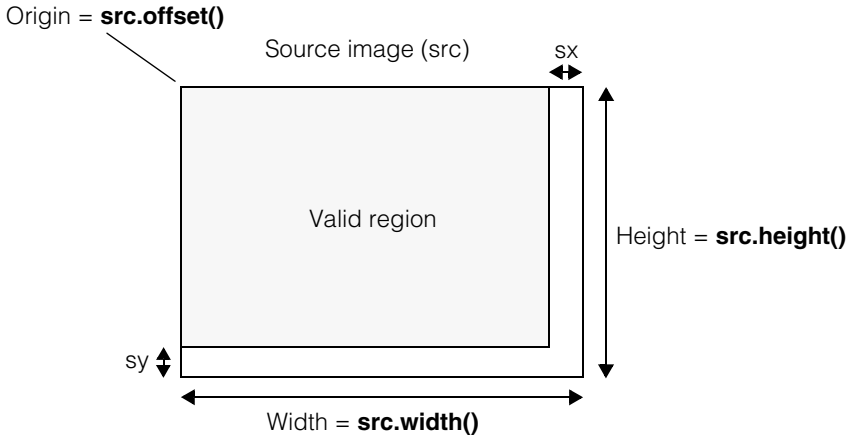
  Parameters

  algorithm The new algorithm.

sx

```
c_UInt32 sx() const;
void sx(c_UInt32 sx);
```

sx and **sy** specify the x and y separation of image samples (in pixels) used by **cfImageSharpness()** to determine image sharpness when the *eAbsDiff* mode is used. See the following diagram where *src* is the source image.



Notes

sx and **sy** are used only with the algorithm *eAbsDiff*.

- `c_UInt32 sx() const;`

  Returns the current value of **sx**.

- `void sx(c_UInt32 sx);`

  Sets a new **sx** value.

  Parameters

  sx The new **sx** value.

sy

```
c_UInt32 sy() const;
void sy(c_UInt32 sy);
```

sx and **sy** specify the x and y separation of image samples (in pixels) used by **cfImageSharpness()** to determine image sharpness when the *eAbsDiff* mode is used.

Notes

sx and **sy** are used only with the algorithm *eAbsDiff*.

- `c_UInt32 sy() const;`

  Returns the current value of **sy**.

- `void sy(c_UInt32 sy);`

  Sets a new **sy** value.

  Parameters

  sy The new **sy** value.

---

freqBand

```
const ccRange& freqBand() const;
void freqBand(const ccRange& freqBand);
```

Image grey scale patterns can be described in terms of frequency where sharp edges in the patterns represent high frequencies, and blurred edges represent a low frequencies. The total frequency range is expressed as 0 through 0.5 (lowest frequency to highest).

When the algorithm is *eBandPass*, this parameter specifies a frequency band used by **cfImageSharpness()** to determine image sharpness. **cfImageSharpness()** uses only frequencies occurring in this band and reports the average of these found frequencies as the image sharpness score.

The default frequency range is 0.25 through 0.45. All images contain some noise which has a very high frequency. Exclude the top of the frequency range (near 0.5) as we have done with the defaults to exclude noise from the sharpness calculation.

Notes

This parameter is used only when the algorithm is *eBandPass*.

- `const ccRange& freqBand() const;`

  Returns the current frequency band.

- `void freqBand(const ccRange& freqBand);`

  Sets a new frequency band.

  Parameters

  freqBand The new frequency band.

Throws

ccImageSharpnessDefs::BadParams
If the low cutoff frequency is less than zero, or if the high cutoff frequency is greater than 0.5.

noiseLevel

```
double noiseLevel() const;

void noiseLevel( double nLevel );
```

Get/Set the strength of the noise present in the image. As the focus is adjusted the image contrast can change. When the contrast is very low sensor noise can dominate, which can result in artificially high sharpness scores. This parameter makes the *eAutoCorrelation* sharpness function more robust to this effect. Ideally you should set this to a value below the contrast of the image at about the sensor noise level. The default is 0.

- `double noiseLevel() const;`

  Returns the current image noise level setting.

- `void noiseLevel( double nLevel );`

  Sets the image noise level.

  Parameters

  nLevel The new image noise level.

  Throws

  ccImageSharpnessDefs::BadParams
  If *nLevel* is < 0.0.

  Notes

  This parameter is used only by algorithm *eAutoCorrelation*

lowPassSmoothing

```
c_Int32 lowPassSmoothing() const;

void lowPassSmoothing(c_Int32 lowPassSmoothing);
```

Get/Set the low-pass smoothing value. The image is processed using Gaussian smoothing before sharpness is measured. The default value is 0 (no smoothing).

- `c_Int32 lowPassSmoothing() const;`

  Returns the current low-pass smoothing value setting.

- `void lowPassSmoothing(c_Int32 lowPassSmoothing);`

  Sets the low-pass smoothing value.

  Parameters

  lowPassSmoothing
  The new low-pass smoothing value.

  Throws

  ccImageSharpnessDefs::BadParams
  If *lowPassSmoothing* is < 0.

  Notes

  For most images, use a low-pass smoothing value of 0 (no smoothing).

  This parameter is used only by algorithm *eGradientEnergy*.

ccStatistics

```
#include <ch_cvl/stats.h>

class ccStatistics;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a population of samples of type **double**. The class include members that allow you to obtain basic statistical measures of the population.

Constructors/Destructors

ccStatistics

```
ccStatistics();

ccStatistics(const cmStd vector<double>& data);
```

- `ccStatistics();`

  Constructs an object with no samples.

- `ccStatistics(const cmStd vector<double>& data);`

  Constructs an object with the side population of samples.

  Parameters

  data The population.

Public Member Functions

reset

```
void reset();
```

Removes all samples from this object.

clear

```
void clear();
```

Removes all samples from this object.

add

```
void add(double val);
void add(const cmStd vector<double> &vals);
void add(const double *vals, c_Int32 numVals);
```

- `void add(double val);`

  Adds a new value to the population. Calling this function increases **n()** by 1.

  Parameters

val	The value to add.

- `void add(const cmStd vector<double> &vals);`

  Adds a vector of new values to the population. Calling this function increases **n()** by *vals.size()*.

  Parameters

vals	The values to add.

- `void add(const double *vals, c_Int32 numVals);`

  Adds the specified range of values from the supplied vector to the population. The values *vals[0]* through *vals[numVals-1]* are added. Calling this function increases **n()** by *numVals*.

  Parameters

vals	The values.
numVals	How many elements of *vals* to add.

n

```
c_Int32 n() const;
```

The number of samples in this population.

mean

```
double mean() const;
```

Returns the mean value of the population. **n()** must be greater than 0.

Throws
ccStatistics::NoSamples
The population has 0 samples.

statMin

```
double statMin() const;
```

Returns the minimum value in the population. **n()** must be greater than 0.

Throws
ccStatistics::NoSamples
The population has 0 samples.

statMax

```
double statMax() const;
```

Returns the maximum value in the population. **n()** must be greater than 0.

Throws
ccStatistics::NoSamples
The population has 0 samples.

sum

```
double sum() const;
```

Returns the sum of the values in the population. **n()** must be greater than 0.

Throws
ccStatistics::NoSamples
The population has 0 samples.

variance

```
double variance() const;
```

Returns the variance of the values in the population. **n()** must be greater than 0.

Throws
ccStatistics::NoSamples
The population has 0 samples.

stdDev

```
double stdDev() const;
```

Returns the standard deviation of the values in the population. **n()** must be greater than 0.

Throws
ccStatistics::NoSamples
The population has 0 samples.

rms `double rms() const;`

Returns the square root of the mean of the squares of the values in the population. **n()** must be greater than 0.

Throws

ccStatistics::NoSamples
The population has 0 samples.

Operators

operator+
```
ccStatistics operator+(const ccStatistics& lhs,
   const ccStatistics& rhs);
```

Adds the two populations and returns the result as a new **ccStatistics** object.

Parameters

lhs The first object to add.

rhs The second object to add.

operator+= `ccStatistics& operator+=(const ccStatistics& s);`

Adds the supplied population to this one.

Parameters

s The population to add.

operator+= `ccStatistics& operator+=(double d);`

Adds the supplied value to this population.

Parameters

d The value to add.

operator== `bool operator==(const ccStatistics& that) const;`

Returns true if the supplied **ccStatistics** object has the same statistical values as this one, false otherwise.

Notes

While the equality operator does not specifically require that the orders of the values in the two populations be the same, as a practical matter if the populations include floating point values, this operator will return true only if the orders of values in the two populations are the same.

Parameters

that	The object to compare to this one.

ccStatistics

ccStdGreyAcqFifo

```
#include <ch_cvl/acq.h>

class ccStdGreyAcqFifo : public ccGreyAcqFifo;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This is a concrete class that describes acquisition FIFO queues for a standard grey-scale camera such as the Sony XC-75.

Notes

There is no need for you to create objects of this type in your program. When you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()** it returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

Constructors/Destructors

ccStdGreyAcqFifo

```
ccStdGreyAcqFifo(const ccVideoFormat& vf,
  ccFrameGrabber& fg);

~ccStdGreyAcqFifo();
```

- ```
  ccStdGreyAcqFifo(const ccVideoFormat& vf,
    ccFrameGrabber& fg);
  ```

  Creates an idle acquisition FIFO suitable for capturing grey-scale images of the format *vf* from the frame grabber *fg*.

  Parameters

vf	The video format of the images to be acquired.
fg	The frame grabber to use to acquire images.
  ```

- `~ccStdGreyAcqFifo();`

  Destructor.

Public Member Functions

properties `virtual ccAnalogAcqProps& properties();`

Returns a reference to the FIFO's properties object.

videoFormat `virtual const ccVideoFormat& videoFormat() const;`

Returns the video format for which this FIFO was constructed.

frameGrabber `virtual ccFrameGrabber& frameGrabber() const;`

Returns the frame grabber for which this FIFO was constructed.

ccStdGreyVideoFormat

```
#include <ch_cvl/vidfmt.h>

class ccStdGreyVideoFormat : public ccGreyVideoFormat;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This is a concrete class that generates acquisition FIFOs of type **ccStdGreyAcqFifo** used by grey-scale cameras.

The acquisition software creates a single object of class **ccStdVideoFormat** (see page 3009) for each camera type and image size. You use these video format objects to create an acquisition FIFO that is appropriate for the camera and frame grabber that your application uses.

To get a video format, you use **ccStdVideoFormat::getFormat()** described on page 3012.

Notes

There is no need for you to use this class directly in your code. Use the **ccStdVideoFormat** class for all of your format programming.

Constructors/Destructors

The acquisition software creates one object for each supported video format. Use the functions listed at the end of the description of **ccStdVideoFormat** on page 3009 to get a reference to the appropriate video format

Public Member Functions

newAcqFifo

```
ccStdGreyAcqFifo* newAcqFifo(ccFrameGrabber& fg) const;
```

Creates and returns a new acquisition FIFO suitable for acquiring images of this format from the frame grabber *fg*.

Parameters

fg	The frame grabber to use to acquire images.

Throws

ccVideoFormat::NotSupported
The specified frame grabber does not support this video format.

ccStdRGB16AcqFifo

```
#include <ch_cvl/acq.h>

class ccStdRGB16AcqFifo : public ccRGB16AcqFifo;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This is a concrete class that describe acquisition FIFO queues for a standard color camera such as the Sony XC-003.

Notes

There is no need for you to create objects of this type in your program. When you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()** it returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

Constructors/Destructors

ccStdRGB16AcqFifo

```
ccStdRGB16AcqFifo (const ccVideoFormat& vf,
  ccFrameGrabber& fg);

~ccStdRGB16AcqFifo();
```

- ```
  ccStdRGB16AcqFifo (const ccVideoFormat& vf,
    ccFrameGrabber& fg);
  ```

  Creates an idle acquisition FIFO suitable for capturing grey-scale images of the format *vf* from the frame grabber *fg*.

  Parameters

vf	The video format of the images to be acquired.
fg	The frame grabber to use to acquire images.
  ```

- `~ccStdRGB16AcqFifo();`

  Destructor.

Public Member Functions

properties `ccAnalogAcqProps& properties();`

Returns a reference to the FIFO's properties object.

videoFormat `virtual const ccVideoFormat& videoFormat() const;`

Returns the video format for which this FIFO was constructed.

frameGrabber `virtual ccFrameGrabber& frameGrabber() const;`

Returns the frame grabber for which this FIFO was constructed.

ccStdRGB32AcqFifo

```
#include <ch_cvl/acq.h>

class ccStdRGB32AcqFifo : public ccRGB32AcqFifo;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

This is a concrete class that describe acquisition FIFO queues for a standard color camera such as the Sony XC-003.

Notes

There is no need for you to create objects of this type in your program. When you create an acquisition FIFO with **ccStdVideoFormat::newAcqFifoEx()** it returns a **ccAcqFifo** base class pointer which you use to access **ccAcqFifo** methods to acquire images.

Constructors/Destructors

ccStdRGB32AcqFifo

```
ccStdRGB32AcqFifo (const ccVideoFormat& vf,
  ccFrameGrabber& fg);

~ccStdRGB32AcqFifo();
```

- ```
  ccStdRGB32AcqFifo (const ccVideoFormat& vf,
    ccFrameGrabber& fg);
  ```

  Creates an idle acquisition FIFO suitable for capturing grey-scale images of the format *vf* from the frame grabber *fg*.

  Parameters

vf	The video format of the images to be acquired.
fg	The frame grabber to use to acquire images.
  ```

- `~ccStdRGB32AcqFifo();`

  Destructor.

Public Member Functions

properties `ccAnalogAcqProps& properties();`

Returns a reference to the FIFO's properties object.

videoFormat `virtual const ccVideoFormat& videoFormat() const;`

Returns the video format for which this FIFO was constructed.

frameGrabber `virtual ccFrameGrabber& frameGrabber() const;`

Returns the frame grabber for which this FIFO was constructed.

ccStdVideoFormat

```
#include <ch_cvl/vidfmt.h>

class ccStdVideoFormat: public ccStdGreyVideoFormat;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This is a concrete class that the acquisition software uses to describe operations on video formats. The acquisition software defines a video format as the pairing of a particular camera type with a particular image size.

The acquisition software creates a single object of type **ccStdVideoFormat** for each camera type and image size. You use these video format objects to create an acquisition FIFO that is appropriate for the camera and frame grabber that your application uses. To get a video format, you call **ccStdVideoFormat::getFormat()** described in this reference page.

Video formats are either built in to the CVL code or are available though Camera Configuration Files (CCFs). Some frame grabbers can use only built-in video formats, while others use CCF-based video formats. For CVL versions prior to 6.0, CCFs are located in the directory specified by the environment variable *COGNEX_CCF_PATH* (usually *C:\vision\ccf)*. For CVL 6.0 and later versions, CCFs are located in *%VISION_ROOT%\bin\win32\cvl*.

Frame grabber/CVM combinations that use CCFs, such as the MVS-8120/CVM1 and the MVS-8120/CVM6 can also use the built-in formats in a compatibility mode. For example, if you specify the video format “Sony XC75 640x480”, CVL will actually use the corresponding CCF for the sync model you specify. If you are using a frame grabber/CVM combination that supports CCFs, use the CCF formats in preference to the built-in formats.

The *CVL Supported Cameras* HTML file lists the camera formats available for all of the frame grabbers supported in the current release. CCF-based formats have names that end in “CCF,” while built-in formats do not. You can also use the function **ccVideoFormat::filterList()** on page 3403 to determine which video formats are available for a given frame grabber.

Constructors/Destructors

ccStdVideoFormat

```
~ccStdVideoFormat();
```

Destructor.

The acquisition software creates one object for each supported video format. Use the functions listed at the end of this class to get a reference to the appropriate video format.

Enumerations

ceRGB16Pack

```
typedef enum { ckRGB16 = 16 } ceRGB16Pack;
```

The single constant for this enumeration, *ckRGB16*, is used as a parameter to **newAcqFifo()** to specify that it should return a 16-bit RGB acquisition FIFO.

ceRGB32Pack

```
typedef enum { ckRGB32 = 32 } ceRGB32Pack;
```

The single constant for this enumeration, *ckRGB32*, is used as a parameter to **newAcqFifo()** to specify that it should return a 32-bit RGB acquisition FIFO.

Public Member Functions

newAcqFifo

```
ccStdGreyAcqFifo* newAcqFifo(ccFrameGrabber& fg) const;

ccStdRGB16AcqFifo* newAcqFifo(ccFrameGrabber& fg,
  ceRGB16Pack packing) const;

ccStdRGB32AcqFifo* newAcqFifo(ccFrameGrabber& fg,
  ceRGB32Pack packing) const;
```

- `ccStdGreyAcqFifo* newAcqFifo(ccFrameGrabber& fg) const;`

  Creates and returns a new acquisition FIFO suitable for acquiring grey-scale from the frame grabber *fg*.

  Parameters

  fg — The frame grabber to use to acquire images.

  Throws

  ccVideoFormat::NotSupported — The specified frame grabber does not support this video format.

- ```
  ccStdRGB16AcqFifo* newAcqFifo(ccFrameGrabber& fg,
    ceRGB16Pack packing) const;
  ```

  Creates and returns a new acquisition FIFO suitable for acquiring 16-bit RGB images from the frame grabber *fg*.

  Parameters

  fg — The frame grabber to use to acquire images.

  packing — The packing scheme of each pixel. For a 16-bit RGB acquisition FIFO, this parameter must be

  ccStdVideoFormat::ckRGB16

  Throws

  ccVideoFormat::NotSupported
  The specified frame grabber does not support this video format.

- ```
  ccStdRGB32AcqFifo* newAcqFifo(ccFrameGrabber& fg,
    ceRGB32Pack packing) const;
  ```

  Creates and returns a new acquisition FIFO suitable for acquiring 32-bit RGB images from the frame grabber *fg*.

  Parameters

  fg — The frame grabber to use to acquire images.

  packing — The packing scheme of each pixel. For a 32-bit RGB acquisition FIFO, this parameter must be

  ccStdVideoFormat::ckRGB32

  Throws

  ccVideoFormat::NotSupported
  The specified frame grabber does not support this video format.

newAcqFifoEx

```
ccAcqFifo* newAcqFifoEx(
  ccFrameGrabber& fg,
  ceImageFormat fmt = ceImageFormat_Unknown) const;
```

Create and return a **ccAcqFifo**. This is the recommended way to create a **ccAcqFifo**.

Parameters

fg — The frame grabber to use to acquire images.

fmt — The image format.

Notes

The image format of a CCF is the format of a camera, while the image format of a **ccAcqFifo** is the format of images acquired into system memory. Use *fmt* = *ceImageFormat_Unknown*, when you intend to acquire images in the native format that the camera outputs. There is no format conversion of any kind during image acquisition.

fullList

```
static const cmStd vector<const ccStdVideoFormat*>&
  fullList();
```

Returns a vector of pointers to all supported **ccVideoFormat** objects.

filterList

```
static cmStd vector<const ccStdVideoFormat*> filterList(
  const cmStd vector<const ccStdVideoFormat*>& list,
  ccFrameGrabber& fg);
```

Returns a vector of pointers to **ccVideoFormat** objects supported by the given **ccVideoFormat** within the given vector of pointers to **ccVideoFormat** objects.

Parameters

list	List of **ccVideoFormat** objects.
fg	The specified frame grabber.

Example:

```
cmStd vector<const ccVideoFormat*> myVidFmts =
  ccVideoFormat::filterList(ccVideoFormat::fullList(), fg);
```

Where *fg* is a particular **ccFrameGrabber**. The function returns the complete list of **ccVideoFormat** objects which are support on frame grabber *fg*.

Static Functions

getFormat

```
static const ccStdVideoFormat& getFormat(
  ccCvlString& name);
```

If the named format is one of the built-in camera formats, or is available through a Camera Configuration File (CCF), this function returns the appropriate **ccStdVideoFormat** object.

Parameters

name	The name of the video format. Must match exactly the string returned by **ccVideoFormat::name()** for a particular video format.

Example

This is how you would get a video format for a Sony XC75 camera:

```
const ccStdVideoFormat& fmt =
        ccStdVideoFormat::getFormat(cmT("Sony XC75 640x480"))
```

Notes

The function **ccVideoFormat::fullList()** on page 3403 returns a vector of pointers to video format objects for all of the supported video formats. The function **ccVideoFormat::filterList()** returns a vector of pointers to all of the video formats supported by a particular frame grabber. Not all video formats are supported on all Cognex frame grabbers. Please consult the *Release Notes* of the specific CVL release you are using for a complete list of supported video formats.

Before CVL 5.5, Cognex recommended using the functions described in the next section to obtain video formats. While these functions are retained for backward compatibility, their use is no longer recommended.

Some cameras or frame grabbers are unable to supply images with the exact dimensions specified in the video format. To determine the actual image size, use the functions **ccPelBuffer::height()** and **ccPelBuffer:width()**. The functions **ccVideoFormat::height()** and **ccVideoFormat::width()** return the nominal size of the image as described in the format names, not the actual image sizes.

Throws

ccVideoFormat::NotFound

name does not match a known format name.

Functions

These functions are retained for compatibility with programs written with earlier versions of CVL. Instead of using these functions to obtain a video format object, you should use **ccStdVideoFormat::getFormat()**.

Each of these functions returns a unique **ccStdVideoFormat** object for the specified video format.

Not all video formats are supported on all Cognex frame grabbers. Please consult the *Release Notes* of the specific CVL release you are using for a complete list of supported video formats.

cfCvc1000_640x480

```
const ccStdVideoFormat& cfCvc1000_640x480();
```

Returns a video format object for a Cognex CVC-1000 high-speed analog camera with a full resolution image size of 640x480.

ccStdVideoFormat

cfKpf100_1280x1024

```
const ccStdVideoFormat& cfKpf100_1280x1024();
```

Returns a video format object for a Hitachi KP-F100 digital camera with a full resolution image size of 1280x1024.

cfLSFth_2048

```
const ccStdVideoFormat& cfLSFth_2048();
```

Returns a video format object for an NED FTH 20 MHz analog line scan camera.

cfLSScd_2048

```
const ccStdVideoFormat& cfLSScd_2048();
```

Returns a video format object for a Mitsubishi SCD-2048 20 MHz analog line scan camera.

cfLSTest_2048()

```
const ccStdVideoFormat& cfLSTest_2048();
```

Returns a video format for a virtual line scan camera used for troubleshooting. Images acquired through this camera consist of eight 256-pixel-wide light-to-dark gradients.

cfTm9700_640x480

```
const ccStdVideoFormat& cfTm9700_640x480();
```

Returns a video format object for a Pulnix TM-9701 RS-170 rapid reset camera with a full resolution image size of 640x480.

cfIk542_640x480

```
const ccStdVideoFormat& cfIk542_640x480();
```

Returns a video format object for a Toshiba IK-542 RS-170 rapid reset camera with a full resolution image size of 640x480.

cfIkm41ma_320x240

```
const ccStdVideoFormat& cfIkm41ma_320x240();
```

Returns a video format object for a Toshiba IK-M41MA RS-170 rapid reset camera with a half resolution image size of 320x240.

cfTm6cn_760x574

```
const ccStdVideoFormat& cfTm6cn_760x574();
```

Returns a video format object for a Pulnix TM-6CN CCIR camera with a full resolution image size of 760x574. The TM-6CN provides synchronization signals to the frame grabber.

cfTm7ex_640x480

```
const ccStdVideoFormat& cfTm7ex_640x480();
```

Returns a video format object for a Pulnix TM-7EX RS-170 camera with a full resolution image size of 640x480.

cfTm7ex_320x240

```
const ccStdVideoFormat& cfTm7ex_320x240();
```

Returns a video format object for a Pulnix TM-7EX RS-170 camera with a half resolution image size of 320x240.

cfTm7ex_640x240

```
const ccStdVideoFormat& cfTm7ex_640x240();
```

Returns a video format object for a Pulnix TM-7EX RS-170 camera with a single field image size of 640x240.

cfXc55_640x480 `const ccStdVideoFormat& cfXc55_640x480();`

Returns a video format object for a Sony XC-55 RS-170 rapid reset camera with a full resolution image size of 640x480.

cfXc75_640x480 `const ccStdVideoFormat& cfXc75_640x480();`

Returns a video format object for a Sony XC-75 RS-170 camera with a full resolution image size of 640x480.

cfXc75_640x240 `const ccStdVideoFormat& cfXc75_640x240();`

Returns a video format object for a Sony XC-75 RS-170 camera with a single field image size of 640x240.

cfXc75_320x240 `const ccStdVideoFormat& cfXc75_320x240();`

Returns a video format object for a Sony XC-75 RS-170 camera with a half resolution image size of 320x240.

cfXc75rr_320x240

```
const ccStdVideoFormat& cfXc75rr_320x240();
```

Returns a video format object for a Sony XC-75 RR RS-170 rapid reset camera with a half resolution image size of 320x240.

cfXc75cerr_380x287

```
const ccStdVideoFormat& cfXc75cerr_380x287();
```

Returns a video format object for a Sony XC-75CE RR CCIR rapid reset camera with a half resolution image size of 380x287.

cfXc75ce_760x574

```
const ccStdVideoFormat& cfXc75ce_760x574();
```

Returns a video format object for a Sony XC-75CE CCIR camera with a full resolution image size of 760x574.

cfXc75ce_760x287

```
const ccStdVideoFormat& cfXc75ce_760x287();
```

Returns a video format object for a Sony XC-75CE CCIR camera with a single field image size of 760x287.

cfXc75ce_380x287

```
const ccStdVideoFormat& cfXc75ce_380x287();
```

Returns a video format object for a Sony XC-75CE CCIR camera with a half resolution image size of 380x287.

cfXc7500_640x480

```
const ccStdVideoFormat& cfXc7500_640x480();
```

Returns a video format object for a Sony XC-7500 dual tap rapid reset RS-170 camera with a full resolution image size of 640x480.

cfXc003_640x480

```
const ccStdVideoFormat& cfXc003_640x480();
```

Returns a video format object for a Sony XC-003 NTSC color camera with a full resolution image size of 640x480.

cfXc003p_760x574

```
const ccStdVideoFormat& cfXc003p_760x574();
```

Returns a video format object for a Sony XC-003P PAL color camera with a full resolution image size of 760x574.

cfDLSScd_2048x2048

```
const ccStdVideoFormat& cfDLSScd_2048x2048();
```

Returns a video format object for a Mitsubishi SCD-2048 40 MHz digital line scan camera with an image size up to 2048 pixels wide and up to 2048 lines high. The default image size is 2048x2048.

Use the ROI property to set the exact dimensions of the image. (See **ccRoiProp** on page 2763.)

cfDLSScd_1024x4096

```
const ccStdVideoFormat& cfDLSScd_1024x4096();
```

Returns a video format object for a Mitsubishi SCD-1024 40 MHz digital line scan camera with an image size up to 1024 pixels wide and up to 4096 lines high. The default image size is 1024x4096.

Use the ROI property to set the exact dimensions of the image. (See **ccRoiProp** on page 2763.)

cfDLSSpy_2048x2048

```
const ccStdVideoFormat& cfDLSSpy_2048x2048();
```

Returns a video format object for a Dalsa Spyder 40 MHz digital line scan camera with an image size up to 2048 pixels wide and up to 2048 lines high. The default image size is 2048x2048.

Use the ROI property to set the exact dimensions of the image. (See **ccRoiProp** on page 2763.)

cfDLSSpy_1024x4096

```
const ccStdVideoFormat& cfDLSSpy_1024x4096();
```

Returns a video format object for a Dalsa Spyder 40 MHz digital line scan camera with an image size up to 1024 pixels wide and up to 4096 lines high. The default image size is 1024x4096.

Use the ROI property to set the exact dimensions of the image. (See **ccRoiProp** on page 2763.)

ccStdVideoFormat

ccStrobeDelayProp

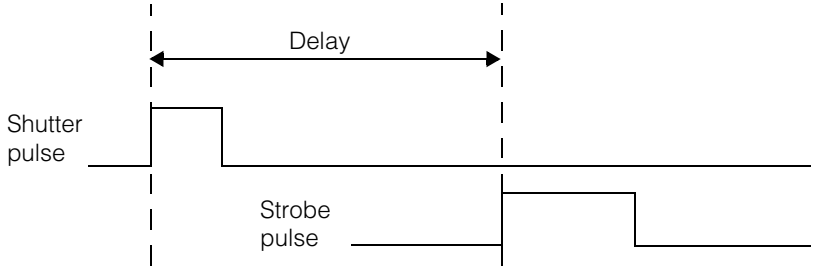
```
#include <ch_cvl/prop.h>

class ccStrobeDelayProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes a property that controls the strobe delay associated with an acquisition FIFO. Strobe delay is the delay time in seconds between the shutter pulse and the strobe firing.



Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccStrobeDelayProp

```
ccStrobeDelayProp();
explicit ccStrobeDelayProp(double delay);
virtual ~ccStrobeDelayProp() {};
```

- `ccStrobeDelayProp();`

  Default constructor. Default strobe delay = 0.

- `explicit ccStrobeDelayProp(double delay);`

  Constructor specifying strobe delay.

  Parameters

  delay Strobe delay in seconds.

- `virtual ~ccStrobeDelayProp() {};`

  Destructor.

Public Member Functions

strobeDelay

```
void strobeDelay(double delay);
double strobeDelay() const;
```

Specifies the strobe delay for acquisitions. The strobe delay is the time in seconds between the shutter pulse and the strobe pulse. The default value is 0.

- `void strobeDelay(double delay);`

  Sets a new strobe delay.

  Parameters

  delay The new strobe delay in seconds.

- `double strobeDelay() const;`

  Returns the current strobe delay.

Constants

defaultStrobeDelay

```
static const double defaultStrobeDelay;
```

The default is 0.

ccStrobeDelayProp

ccStrobeProp

```
#include <ch_cvl/prop.h>

class ccStrobeProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes the strobe property of an acquisition FIFO queue. The strobe property determines whether an acquisition uses a strobe or ambient lighting.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccStrobeProp

```
ccStrobeProp();

explicit ccStrobeProp(bool enabled);
```

- `ccStrobeProp();`

  Create a new strobe property not associated with any FIFO. Strobing is initially disabled.

- `explicit ccStrobeProp(bool enabled);`

  Create a new strobe property not associated with any FIFO. Strobing is initially enabled or disabled as specified.

  Parameters

  enabled — True if strobing is enabled; false if acquisitions use ambient lighting.

Public Member Functions

strobeEnable

```
bool strobeEnable() const;
void strobeEnable(bool enable);
```

- `bool strobeEnable() const;`

  Returns true if strobing is enabled or false if acquisition uses ambient lighting.

- `void strobeEnable(bool enable);`

  Enable or disable strobing.

  Parameters

  enabled — True if strobing is enabled, false if acquisitions use ambient lighting.

strobePulseDuration

```
double strobePulseDuration() const;
void strobePulseDuration(double seconds);
```

The strobe pulse duration is the pulse width used to fire the strobe. It is not necessarily the duration of the strobe flash.

- `double strobePulseDuration() const;`

  Return the strobe output line duration.

- `void strobePulseDuration(double seconds);`

  Set the strobe pulse duration. The strobe pulse is used to initiate a strobe flash. Check your strobe specifications to determine the strobe pulse width requirements.

  Parameters

  seconds — The width of the strobe pulse in seconds. A value of 0 means to use the shortest possible strobe pulse width. On most platforms, pulse width has a granularity between one millisecond and one microsecond.

  Throws

  ccStrobeProp::BadParams
  seconds was less than 0.

Notes

strobePulseDuration() applies to the MVS-8100L and frame grabbers that support CCF style video formats, including the MVS-8100M+, MVS-8120, and the MVS-8500 series.

strobePulseDuration() has no effect on MVS-8100M, MVS-8100C, and MVS-8100C/CPCI.

strobeHigh

```
bool strobeHigh() const;
void strobeHigh(bool polarity);
```

- `double strobeHigh() const;`

Returns the polarity of the strobe pulse.

- `void strobeHigh(bool polarity);`

Sets the polarity of the strobe pulse. The default value is true (high polarity). Consult your hardware's documentation for the appropriate setting of strobe polarity.

Parameters

polarity	If true, set the active pulse to high. If false, sets the active pulse to low.

ccStrobeProp

ccSymbologyParamsUPCEAN

```
#include <ch_cvl/id.h>

class ccSymbologyParamsUPCEAN;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the UPC/EAN symbology.

Enumerations

UPCEANType

```
enum UPCEANType;
```

This enumeration defines the UPC/EAN symbol types. The values in this enumeration are returned by **ccIDDecodeResult::symbologySubtype()**.

Value	meaning
eUPC_A	UPC-A.
eUPC_E	UPC-E
eEAN_8	EAN-8
eEAN_13	EAN-13
eAddOn5	5-digit EAN add-on
eAddOn2	2-digit EAN add-on

Constructors/Destructors

ccSymbologyParamsUPCEAN

```
ccSymbologyParamsUPCEAN();
```

Constructs an object with the following values:.

isAddOnEnabled: true
isEAN8AddOnValid: false
isUPCE1Enabled: false
isUPCEExpanded: true

Public Member Functions

isAddOnEnabled

```
bool isAddOnEnabled() const;

void isAddOnEnabled(bool isAddOnEnabled);
```

- `bool isAddOnEnabled() const;`

  Returns true if reading of UPC/EAN symbols with 2- or 5-digit add-on component is enabled, false otherwise.

- `void isAddOnEnabled(bool isAddOnEnabled);`

  Determines whether the reading of UPC/EAN symbols with 2- or 5-digit add-on component is enabled.

  Parameters

  isAddOnEnabled

  Specify true to enable reading of add-on components, false to disable reading of add-on components.

isUPCE1Enabled

```
bool isUPCE1Enabled() const;

void isUPCE1Enabled(bool isUPCE1Enabled);
```

- `bool isUPCE1Enabled() const;`

  Returns true if reading UPC-E symbols is enabled, false otherwise.

- `void isUPCE1Enabled(bool isUPCE1Enabled);`

  Sets whether reading UPC-E symbols is enabled.

  Parameters

  isUPCE1Enabled

  Specify true to enable reading UPC-E symbols, false to disable it.

  Notes

  UPC-E is applicable only to number system 0, and is referred as UPC-E0. UPC-E1 is a non-standard variation of UPC-E with number system 1. It is created by inverting the parity pattern for the implicit modulo check character. When isUPCE1Enabled is set to true, all UPC-E parameters apply to both UPC-E0 and UPC-E1 symbols.

isUPCEExpanded

```
bool isUPCEExpanded() const;

void isUPCEExpanded(bool isUPCEExpanded);
```

- `bool isUPCEExpanded() const;`

  Returns true if the expansion of decoded UPC-E strings to the 13-digit EAN-13 representation is enabled, false if it is disabled.

- `void isUPCEExpanded(bool isUPCEExpanded);`

  Sets the expansion of decoded UPC-E strings to the 13-digit EAN-13 representation.

  Parameters

  isUPCEExpanded

  Specify true to enable expansion, false to disable it.

  Notes

  When this flag is set to false, the decoded string of a UPC-E symbol is returned as <number system> + 6 digits + implicit check. When this flag is set to true, the decoded string is returned in 13 digit EAN-13 format.

  The standard UPC-E symbol identifier is used regardless of whether UPC-E is expanded or not. Since it can not indicate a UPC-E string transmitted in non-expanded format, number of bytes in the result's decoded string shall be used to determine this information.

isEAN8AddOnValid

```
bool isEAN8AddOnValid() const;

void isEAN8AddOnValid(bool isEAN8AddOnValid);
```

- `bool isEAN8AddOnValid() const;`

  Returns true if the tool will treat EAN-8 symbols with 2- or 5-digit add-on symbols as valid, false otherwise.

- `void isEAN8AddOnValid(bool isEAN8AddOnValid);`

  Sets whether or not the tool will treat EAN-8 symbols with 2- or 5-digit add-on symbols as valid, false otherwise.

  Parameters
  isEAN8AddOnValid

  Notes

  A standard EAN-8 symbol is not allowed to have add-on symbols. Enabling this flag allows reading of non-standard EAN-8 symbols where an add-on symbol is used.

  The decoded EAN-8 string is returned aa 8 digits (no add-on), 10 digits (2 digit add-on), or 13 digits (5-digit add-on).

  Notes
  This flag has no effect when **isAddOnEnabled** is set to false.

Operators

operator==

```
bool operator== (const ccSymbologyParamsUPCEAN& that)
const;
```

Returns true if the supplied object is equal to this one.

Parameters

that	The object to compare to this one.

Notes
Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsCodabar

```
#include <ch_cvl/id.h>

class ccSymbologyParamsCodabar;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the Codabar symbology.

Constructors/Destructors

ccSymbologyParamsCodabar

```
ccSymbologyParamsCodabar();
```

Constructs an object with the following values:

transmitStartStop: false
hasCheckChar: false
transmitCheckChar: false
lengthMin: 3
lengthMax: 40

Public Member Functions

transmitStartStop

```
bool transmitStartStop() const;

void transmitStartStop(bool transmitStartStop);
```

- `bool transmitStartStop() const;`

  Returns true if the tool includes start and stop characters in the returned decoded string, false otherwise.

- `void transmitStartStop(bool transmitStartStop);`

  Sets whether the tool includes start and stop characters in the returned decoded string.

Parameters

transmitStartStop

Specify true to include start and stop characters, false to not include them.

hasCheckChar

```
bool hasCheckChar() const;
void hasCheckChar(bool hasCheckChar);
```

- `bool hasCheckChar() const;`

Returns true if the tool requires that the symbol include a valid embedded checksum character, false if no checksum character is expected.

- `void hasCheckChar(bool hasCheckChar);`

Sets whether the tool requires that the symbol include a valid embedded checksum character.

Parameters

hasCheckChar Specify true to require an embedded checksum character, false to not require a checksum character.

Notes

When this flag is set to true, the checksum is validated before the result is reported.

transmitCheckChar

```
bool transmitCheckChar() const;
void transmitCheckChar(bool transmitCheckChar);
```

- `bool transmitCheckChar() const;`

Returns true if the tool includes the embedded checksum character in the returned decoded string, false if the tool does not include the checksum character.

- `void transmitCheckChar(bool transmitCheckChar);`

Sets whether the tool includes the embedded checksum character in the returned decoded string.

Parameters

transmitCheckChar

Specify true to have the tool include the checksum character.

Notes

If **hasCheckChar()** has a value of false, this property has no effect on the tool.

lengthMin

```
c_Int32 lengthMin() const;
```

Returns the minimum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes fewer bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

lengthMax

```
c_Int32 lengthMax() const;
```

Returns the maximum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes more bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

setLengthRange

```
void setLengthRange(c_Int32 lengthMin, c_Int32 lengthMax);
```

Sets the minimum and maximum number of bytes that are expected to be encoded in the symbol. If the number of bytes in the decoded symbol is outside of the specified range, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

Parameters

lengthMin The minimum number of bytes.

lengthMax The maximum number of bytes.

Throws

ccIDDefs::BadParams

lengthMin is less than *ccIDDefs::kLinearLengthMin*, *lengthMax* is greater than *ccIDDefs::kLinearLengthMax*, or *lengthMin* is greater than *lengthMax*.

Operators

operator==

```
bool operator== (const ccSymbologyParamsCodabar& that)
   const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsCode39

```
#include <ch_cvl/id.h>

class ccSymbologyParamsCode39;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the Code 39 symbology.

Constructors/Destructors

ccSymbologyParamsCode39

```
ccSymbologyParamsCode39();
```

Constructs an object with the following values:

fullASCIIMode: false
hasCheckChar: false
transmitCheckChar: false
lengthMin: 3
lengthMax: 40

Public Member Functions

fullASCIIMode

```
bool fullASCIIMode() const;

void fullASCIIMode(bool fullASCIIMode);
```

- ```
  bool fullASCIIMode() const;
  ```

  Returns true if the tool interprets the decoded string as full ASCII characters, false if only the 44-character subset is interpreted (A-Z; 0-9; *-$% ./+).
  ```

- `void fullASCIIMode(bool fullASCIIMode);`

  Gets/sets fullASCIIMode flag.

  Determines whether the tool interprets the decoded string as full ASCII characters or only the 44-character subset (A-Z; 0-9; *-$% ./+).

  Parameters

  fullASCIIMode Specify true to interpret full ASCII, false for the 44-character subset.

hasCheckChar

```
bool hasCheckChar() const;
void hasCheckChar(bool hasCheckChar);
```

- `bool hasCheckChar() const;`

  Returns true if the tool requires that the symbol include a valid embedded checksum character, false if no checksum character is expected.

- `void hasCheckChar(bool hasCheckChar);`

  Sets whether the tool requires that the symbol include a valid embedded checksum character.

  Parameters

  hasCheckChar Specify true to require an embedded checksum character, false to not require a checksum character.

  Notes

  When this flag is set to true, the checksum is validated before the result is reported.

transmitCheckChar

```
bool transmitCheckChar() const;
void transmitCheckChar(bool transmitCheckChar);
```

- `bool transmitCheckChar() const;`

  Returns true if the tool includes the embedded checksum character in the returned decoded string, false if the tool does not include the checksum character.

- `void transmitCheckChar(bool transmitCheckChar);`

  Sets whether the tool includes the embedded checksum character in the returned decoded string.

  Parameters

  transmitCheckChar

  Specify true to have the tool include the checksum character.

  Notes

  If **hasCheckChar()** has a value of false, this property has no effect on the tool.

lengthMin `c_Int32 lengthMin() const;`

Returns the minimum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes fewer bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

lengthMax `c_Int32 lengthMax() const;`

Returns the maximum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes more bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

setLengthRange

`void setLengthRange(c_Int32 lengthMin, c_Int32 lengthMax);`

Sets the minimum and maximum number of bytes that are expected to be encoded in the symbol. If the number of bytes in the decoded symbol is outside of the specified range, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

Parameters

lengthMin The minimum number of bytes.

lengthMax The maximum number of bytes.

Throws

ccIDDefs::BadParams

lengthMin is less than *ccIDDefs::kLinearLengthMin*, *lengthMax* is greater than *ccIDDefs::kLinearLengthMax*, or *lengthMin* is greater than *lengthMax*.

Operators

operator==

```
bool operator== (const ccSymbologyParamsCode39& that)
  const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsCode93

```
#include <ch_cvl/id.h>

class ccSymbologyParamsCode93;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the Code 93 symbology.

Constructors/Destructors

ccSymbologyParamsCode93

```
ccSymbologyParamsCode93();
```

Constructs an object with the following values:.

lengthMin: 3
lengthMax: 40

Public Member Functions

lengthMin

```
c_Int32 lengthMin() const;
```

Returns the minimum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes fewer bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

lengthMax

```
c_Int32 lengthMax() const;
```

Returns the maximum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes more bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

setLengthRange

```
void setLengthRange(c_Int32 lengthMin, c_Int32 lengthMax);
```

Sets the minimum and maximum number of bytes that are expected to be encoded in the symbol. If the number of bytes in the decoded symbol is outside of the specified range, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

Parameters

lengthMin The minimum number of bytes.

lengthMax The maximum number of bytes.

Throws

ccIDDefs::BadParams

lengthMin is less than *ccIDDefs::kLinearLengthMin*, *lengthMax* is greater than *ccIDDefs::kLinearLengthMax*, or *lengthMin* is greater than *lengthMax*.

Operators

operator==

```
bool operator== (const ccSymbologyParamsCode93& that)
   const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsCode128

```
#include <ch_cvl/id.h>

class ccSymbologyParamsCode128;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the Code 128 symbology.

Constructors/Destructors

ccSymbologyParamsCode128

```
ccSymbologyParamsCode128();
```

Constructs an object with the following values:.

lengthMin: 3
lengthMax: 40

Public Member Functions

lengthMin

```
c_Int32 lengthMin() const;
```

Returns the minimum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes fewer bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

lengthMax

```
c_Int32 lengthMax() const;
```

Returns the maximum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes more bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

setLengthRange

```
void setLengthRange(c_Int32 lengthMin, c_Int32 lengthMax);
```

Sets the minimum and maximum number of bytes that are expected to be encoded in the symbol. If the number of bytes in the decoded symbol is outside of the specified range, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

Parameters

lengthMin The minimum number of bytes.

lengthMax The maximum number of bytes.

Throws

ccIDDefs::BadParams

lengthMin is less than *ccIDDefs::kLinearLengthMin*, *lengthMax* is greater than *ccIDDefs::kLinearLengthMax*, or *lengthMin* is greater than *lengthMax*.

Operators

operator==

```
bool operator== (const ccSymbologyParamsCode128& that)
   const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsComposite

```
#include <ch_cvl/id.h>

class ccSymbologyParamsComposite;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for composite symbols.

Constructors/Destructors

ccSymbologyParamsComposite

```
ccSymbologyParamsComposite();
```

Constructs an object with the following values:

type: *ccSymbologyParamsComposite::kDefaultComponent2DType*
combineResults: false
transmissionMode: *ccSymbologyParamsComposite::kDefaultDataTransmissionMode*

Enumerations

Component2DType

```
enum Component2DType;
```

This enumeration defines the component symbology types supported by the ID tool. The values in this enumeration are returned by **ccIDDecodeResult::symbologySubtype()**.

Value	meaning
eComponent2DTypeCCA	CC-A
eComponent2DTypeCCB	CC-B
kDefaultComponent2DType	The default type, as defined in id.h.

DataTransmissionMode

```
enum DataTransmissionMode;
```

This enumeration defines the data transmission modes supported for composite symbols.

Value	meaning
eDataTransmissionModeStandard	Both the linear and 2D components of a composite symbol are returned.
eDataTransmissionModeLinearOnly	Only the linear component of a composite symbol is returned.
kDefaultDataTransmissionMode	The default mode, as defined in id.h.

Public Member Functions

type2D

```
c_UInt32 type2D() const;

void type2D(c_UInt32 type);
```

- `c_UInt32 type2D() const;`

  Returns the type of 2D component that the tool will decode. The returned value is formed by ORing together one or more of the values defined in the **ccSymbologyParamsComposite::Component2DType** enumeration.

- `void type2D(c_UInt32 type);`

  Sets the type of 2D component that the tool will decode.

  Parameters

  type The 2D types to decode. This value must be formed by ORing together one or more of the following values:

  ccSymbologyParamsComposite::eDataTransmissionModeStandard
  ccSymbologyParamsComposite::eDataTransmissionModeLinearOnly

  Throws

  ccIDDefs::BadParams

  type is not the result of ORing together one or more values from the **ccSymbologyParamsComposite::Component2DType** enumeration.

combineResults

```
bool combineResults() const;

void combineResults(bool combineResults);
```

- `bool combineResults() const;`

  If true, the tool combines the two decoded components of a composite symbol and returns a single result. If false, the tool reports separate subresuilts.

- `void combineResults(bool combineResults);`

  Controls whether the tool combines the two decoded components of a composite symbol and returns a single result or reports separate subresuilts for the components.

  If the tool combines results, the **decodedString()** and **decodedElementString()** values are concatenated.

  Parameters

  combineResults Specify true to combine results, false to report separate results.

  Notes

  This flag has no effect if the **dataTransmissionMode** is set to *ccSymbologyParamsComposite::eDataTransmissionModeLinearOnly*.

dataTransmissionMode

```
DataTransmissionMode dataTransmissionMode() const;

void dataTransmissionMode(DataTransmissionMode mode);
```

- `DataTransmissionMode dataTransmissionMode() const;`

  Returns the current data transmission mode. The transmission mode controls which results the tool returns.

  If the mode is *ccSymbologyParamsComposite::eDataTransmissionModeLinearOnly*, then only the linear component is returned. If the mode is *ccSymbologyParamsComposite::eDataTransmissionModeLinearOnly*, then both components are returned. **combineResults()** determines how the results are returned.

- `void dataTransmissionMode(DataTransmissionMode mode);`

  Sets the current data transmission mode. The transmission mode controls which results the tool returns.

  If you specify *ccSymbologyParamsComposite::eDataTransmissionModeLinearOnly*, then only the linear component is returned. If you specify *ccSymbologyParamsComposite::eDataTransmissionModeLinearOnly*, then both components are returned. **combineResults()** determines how the results are returned.

  Parameters

  mode The transmission mode to set.

  Throws

  ccIDDefs::BadParams
  mode is invalid.

Operators

operator==

```
bool operator== (
   const ccSymbologyParamsComposite& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsI2of5

```
#include <ch_cvl/id.h>

class ccSymbologyParamsI2of5;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the 2 of 5 symbology.

Constructors/Destructors

ccSymbologyParamsI2of5

```
ccSymbologyParamsI2of5();
```

Constructs an object with the following values:

hasCheckChar: false
transmitCheckChar: false
lengthMin: 6
lengthMax: 40

Public Member Functions

hasCheckChar

```
bool hasCheckChar() const;

void hasCheckChar(bool hasCheckChar);
```

- `bool hasCheckChar() const;`

  Returns true if the tool requires that the symbol include a valid embedded checksum character, false if no checksum character is expected.

- `void hasCheckChar(bool hasCheckChar);`

  Sets whether the tool requires that the symbol include a valid embedded checksum character.

Parameters

hasCheckChar Specify true to require an embedded checksum character, false to not require a checksum character.

Notes

When this flag is set to true, the checksum is validated before the result is reported.

transmitCheckChar

```
bool transmitCheckChar() const;

void transmitCheckChar(bool transmitCheckChar);
```

- `bool transmitCheckChar() const;`

  Returns true if the tool includes the embedded checksum character in the returned decoded string, false if the tool does not include the checksum character.

- `void transmitCheckChar(bool transmitCheckChar);`

  Sets whether the tool includes the embedded checksum character in the returned decoded string.

  Parameters

  transmitCheckChar
  Specify true to have the tool include the checksum character.

  Notes

  If **hasCheckChar()** has a value of false, this property has no effect on the tool.

lengthMin `c_Int32 lengthMin() const;`

Returns the minimum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes fewer bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

lengthMax `c_Int32 lengthMax() const;`

Returns the maximum number of bytes that are expected to be encoded in the symbol. If the decoded symbol includes more bytes, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

setLengthRange

```
void setLengthRange(c_Int32 lengthMin, c_Int32 lengthMax);
```

Sets the minimum and maximum number of bytes that are expected to be encoded in the symbol. If the number of bytes in the decoded symbol is outside of the specified range, **ccIDResult::isDecoded()** returns false and **ccIDResult::failureCode()** is set to *ccIDDefs::eFailureUnexpectedStringLength*.

Parameters

lengthMin The minimum number of bytes.

lengthMax The maximum number of bytes.

Throws

ccIDDefs::BadParams

lengthMin is less than *ccIDDefs::kLinearLengthMin*, *lengthMax* is greater than *ccIDDefs::kLinearLengthMax*, or *lengthMin* is greater than *lengthMax*.

Operators

operator==

```
bool operator== (const ccSymbologyParamsI2of5& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsI2of5

ccSymbologyParamsPDF417

```
#include <ch_cvl/id.h>

class ccSymbologyParamsPDF417;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the PDF417 symbology.

Enumerations

PDF417Type

```
enum PDF417Type;
```

This enumeration defines the PDF417 symbol types. The values in this enumeration are returned by **cclIDDecodeResult::symbologySubtype()**.

Value	meaning
ePDF417	PDF417 code
eMicroPDF417	Micro-PDF417 code
kDefaultPDFType	The default type, as defined in id.h

Constructors/Destructors

ccSymbologyParamsPDF417

```
ccSymbologyParamsPDF417();
```

Constructs an object with the following values:

type: *ccSymbologyParamsPDF417::kDefaultPDFType*

Public Member Functions

type

```
c_UInt32 type() const;

void type(c_UInt32 type);
```

- `c_UInt32 type() const;`

  Returns the PDF417 subtype that the tool will decode. The returned value will be formed by ORing together 1 or more of the following values:

  ccSymbologyParamsPDF417::ePDF417
  ccSymbologyParamsPDF417::eMicroPDF417

- `void type(c_UInt32 type);`

  Sets the PDF417 subtypes that the tool will decode.

  type — The types to decode. *type* must be formed by ORing together 1 or more of the following values:

  ccSymbologyParamsPDF417::ePDF417
  ccSymbologyParamsPDF417::eMicroPDF417

  Throws

  ccIDDefs::BadParams

  type is not the result of ORing together one or more of the values from the not set to a valid bitwise combination of any of the **ccSymbologyParamsPDF417::PDF417Type** enumeration values.

Operators

operator==

```
bool operator== (const ccSymbologyParamsPDF417& that)
   const;
```

Returns true if the supplied object is equal to this one.

Parameters

that — The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsPostal

```
#include <ch_cvl/id.h>

class ccSymbologyParamsPostal;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the Postal symbologies.

Enumerations

PostalType

```
enum PostalType;
```

This enumeration defines the Postal symbol subtypes. The values in this enumeration are returned by **ccIDDecodeResult::symbologySubtype()**.

Value	meaning
ePostal4StateJapanPost	4-state Japan Post code
ePostal4StateAustraliaPost	4-state Australia Post code
ePostal4StateUPU	4-state UPU code
ePostal4StateUSPS	4-state USPS code
ePostalUSPostnet	Postnet code
ePostalUSPlanet	Planet code
ePostalCepnet	CEPNet code
kDefaultPostalType	The default codes, as defined in id.h

Constructors/Destructors

ccSymbologyParamsPostal

```
ccSymbologyParamsPostal();
```

Constructs an object with the following values:

type: *ccSymbologyParamsPostal::kDefaultPostalType*
transmitCheckChar: *false*

Public Member Functions

type

```
c_UInt32 type() const;

void type(c_UInt32 type);
```

- `c_UInt32 type() const;`

  Returns the Postal subtypes that this tool is configured to decode. The returned value is formed by ORing together one or more of the following values:

  ccSymbologyParamsPostal::ePostal4StateJapanPost
  ccSymbologyParamsPostal::ePostal4StateAustraliaPost
  ccSymbologyParamsPostal::ePostal4StateUPU
  ccSymbologyParamsPostal::ePostal4StateUSPS
  ccSymbologyParamsPostal::ePostalUSPostnet
  ccSymbologyParamsPostal::ePostalUSPlanet
  ccSymbologyParamsPostal::ePostalCepnet

- `void type(c_UInt32 type);`

  Sets the Postal subtypes that the tool will decode.

  Parameters

  type — The subtypes to decode. *type* must be formed by ORing together one or more of the following values:

  ccSymbologyParamsPostal::ePostal4StateJapanPost
  ccSymbologyParamsPostal::ePostal4StateAustraliaPost
  ccSymbologyParamsPostal::ePostal4StateUPU
  ccSymbologyParamsPostal::ePostal4StateUSPS
  ccSymbologyParamsPostal::ePostalUSPostnet
  ccSymbologyParamsPostal::ePostalUSPlanet
  ccSymbologyParamsPostal::ePostalCepnet

Throws

ccIDDefs::BadParams

type is not formed by ORing together one or more values from the **ccSymbologyParamsPostal::PostalType** enumeration.

transmitCheckChar

```
bool transmitCheckChar(PostalType type) const;

void transmitCheckChar(PostalType type,
   bool transmitCheckChar);
```

- `bool transmitCheckChar(PostalType type) const;`

Returns true if the tool includes the embedded checksum character in the returned decoded string for the specified symbology subtype, false if the tool does not include the checksum character.

Parameters

type The Postal code type for which to obtain the checksum flag. *type* must be one of the following values:

ccSymbologyParamsPostal::ePostalUSPostnet
ccSymbologyParamsPostal::ePostalUSPlanet

For CEPNet, use *ccSymbologyParamsPostal::ePostalUSPostnet*

- `void transmitCheckChar(PostalType type, bool transmitCheckChar);`

Sets whether the tool includes the embedded checksum character in the returned decoded string for the supplied symbology subtype.

Parameters

type The Postal code type for which to set the checksum flag. *type* must be one of the following values:

ccSymbologyParamsPostal::ePostalUSPostnet
ccSymbologyParamsPostal::ePostalUSPlanet

For CEPNet, use *ccSymbologyParamsPostal::ePostalUSPostnet*

transmitCheckChar

Specify true to return the checksum, false to not return the checksum.

Throws

ccIDDefs::BadParams

type is not *ccSymbologyParamsPostal::ePostalUSPostnet* or *ccSymbologyParamsPostal::ePostalUSPlanet*.

decodePostnetAsCepnet

```
bool decodePostnetAsCepnet() const;

void decodePostnetAsCepnet(bool value);
```

- `bool decodePostnetAsCepnet() const;`

  Returns the flag whether to enable CEPNet decoding.

- `void decodePostnetAsCepnet(bool value);`

  Sets the flag whether to enable CEPNet decoding.

  Parameters

  value Whether to enable CEPNet decoding.

  The default is false.

  Notes

  CEPNet and Postnet use the same symbol character set and checksum algorithm but the lengths are different. When this flag is set to true, the US-Postnet codes will be treated as CEPNet codes.

Operators

operator==

```
bool operator== (const ccSymbologyParamsPostal& that)
   const;
```

Returns true if the supplied object is equal to this one.

Parameters

that The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsRSS

```
#include <ch_cvl/id.h>

class ccSymbologyParamsRSS;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class containing the symbology-specific parameters for the RSS symbology.

Enumerations

RSSType

```
enum RSSType;
```

This enumeration defines the RSS symbol subtypes. The values in this enumeration are returned by **ccIDDecodeResult::symbologySubtype()**.

Value	meaning
eRSS14	RSS - 14
eRSSLimited	RSS - Limited
kDefaultRSSType	The default type, as defined in id.h

Constructors/Destructors

ccSymbologyParamsRSS

```
ccSymbologyParamsRSS();
```

Constructs an object with the following values:

type: *ccSymbologyParamsRSS::kDefaultRSSType*

Public Member Functions

type

```
c_UInt32 type() const;

void type(c_UInt32 type);
```

- `c_UInt32 type() const;`

  Returns the RSS subtypes that this tool is configured to decode. The returned value is formed by ORing together one or more of the following values:

  ccSymbologyParamsRSS::eRSS14
  ccSymbologyParamsRSS::eRSSLimited

- `void type(c_UInt32 type);`

  Sets the RSS subtypes that the tool will decode.

  Parameters

  type — The subtypes to decode. *type* must be formed by ORing together one or more of the following values:

  ccSymbologyParamsRSS::eRSS14
  ccSymbologyParamsRSS::eRSSLimited

  Notes

  This must be a bitwise combination of any of the

  RSSType enum values.

  Throws

  ccIDDefs::BadParams
  type is not formed by ORing together one or more values from the **ccSymbologyParamsRSS::RSSType** enumeration.

Operators

operator==

```
bool operator== (const ccSymbologyParamsRSS& that) const;
```

Returns true if the supplied object is equal to this one.

Parameters

that — The object to compare to this one.

Notes

Two objects are considered equal if all their corresponding data members are exactly equal.

ccSymbologyParamsRSS

ccSynFont

```
#include <ch_cvl/synfont.h>

class ccSynFont : public virtual ccRepBase,
                  public virtual ccPersistent
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

The **ccSynFont** class represents a font, which is a specification for a set of characters. A font can be used to generate rendered images of individual characters or strings, as well as to generate outline shapes and metrics.

A font can be imported either from a set of character bitmaps (a **ccImageFont**) or from a font file, in any one of a variety of supported font file formats.

Constructors/Destructors

ccSynFont

```
ccSynFont();

ccSynFont(const ccSynFont& rhs);

~ccSynFont();
```

- `ccSynFont();`

  Constructs an unimported object.

- `ccSynFont(const ccSynFont& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

- `~ccSynFont();`

  Destructor.

Enumerations

FontType `enum FontType;`

Types of fonts.

The type of a font indicates a broad category of how fonts are internally represented.

Value	Meaning
eNone	No font type (note: not currently used)
eOutline	Outline font (for example, TrueType)
eRaster	Raster font (for example, VideoJet, Markem, Imaje, Zebra, Xymark)
eStroke	Stroke font (for example, Xymark, Domino)
eImage	Image font (character bitmaps; for example, **ccImageFont**)

Operators

operator= `ccSynFont& operator=(const ccSynFont& rhs);`

Assignment operator.

Parameters

rhs The assignment source.

operator== `bool operator==(const ccSynFont& other) const;`

Returns true if and only if this object is equal to the specified other object.

Parameters

other The object with which to compare for equality.

Public Member Functions

import

```
void import(const ccCvlString& filename,
  const ccCvlString& fontname = ccCvlString());

void import(const ccImageFont& imageFont);
```

- ```
  void import(const ccCvlString& filename,
    const ccCvlString& fontname = ccCvlString());
  ```

  Load font from file. Resets all settable values such as *spotSize* and *leading*.

  Parameters

  filename The name of the file.

- ```
  void import(const ccImageFont& imageFont);
  ```

  Load font from the image font. Resets all settable values such as *spotSize* and *leading*.

  Parameters

  imageFont The image font.

  Notes

  A font that is imported this way cannot currently be saved by archiving.

isImported

```
bool isImported() const;
```

Returns whether the font has been imported.

All other functions will throw **ccSynFontDefs::NotImported** if the font has not been imported.

name

```
ccCvlString name() const;
```

Gets the name of the font.

usesSpotSize

```
bool usesSpotSize() const;
```

Returns whether the font is a type that uses *spotSize*, that is, *eRaster* or *eStroke*.

spotSize

```
double spotSize() const;

void spotSize(double diam);
```

- `double spotSize() const;`

  Gets the spot size (diameter) of the laser.

- `void spotSize(double diam);`

  Sets the spot size (diameter) of the laser.

  Parameters

  diam The spot size of the laser in diameter.

  Throws

  ccSynFontDefs::NotImplemented
  usesSpotSize is false.

  ccSynFontDefs::BadParams
  diam is negative.

usesSpotSpacing

```
bool usesSpotSpacing() const;
```

Returns whether the font is a type that is affected by *spotSpacingXScale* and *spotSpacingYScale* in the params.

Notes

Typically, spot and stroke fonts can use spot spacing, while outline fonts cannot.

leading

```
cc2Vect leading(const ccSynFontCharRenderParams&
  charParams = ccSynFontCharRenderParams()) const;

void leading(const cc2Vect& leading);
```

- `cc2Vect leading(const ccSynFontCharRenderParams& charParams = ccSynFontCharRenderParams()) const;`

  Gets the leading, which is the origin-to-origin displacement between the first characters of lines.

  Parameters

  charParams The character rendering parameters.

- `void leading(const cc2Vect& leading);`

  Sets the leading, which is the origin-to-origin displacement between the first characters of lines.

  Typically, **leading.x()** is 0 and **leading.y()** is a positive number.

  Parameters
  leading The new leading value.

  Notes

  Leading is pronounced "ledding."

isKnown `bool isKnown(c_Int32 charCode) const;`

Returns whether or not the given character code is recognized by this font.

Parameters
charCode The given character code.

areKnown

```
bool areKnown(const cmStd vector<c_Int32>& chars) const;
bool areKnown(const ccCvlString& str) const;
```

- `bool areKnown(const cmStd vector<c_Int32>& chars) const;`

  Returns whether or not all the given character codes are recognized by this font.

  Parameters
  chars The given character codes.

- `bool areKnown(const ccCvlString& str) const;`

  Returns whether or not all the given character codes are recognized by this font.

  Parameters
  str The given character codes.

isBlank `bool isBlank(c_Int32 key) const;`

Returns whether or not the given character is a blank (space).

Parameters
key The given character.

Throws

ccSynFontDefs::UnknownChar
key is not in the font.

cellRect

```
ccRect cellRect(c_Int32 charCode,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;
```

Gets the cell rectangle of one character.

Parameters

charCode The code of the character.

charParams The character rendering parameters.

encloseCellRect

```
ccRect encloseCellRect(const cmStd vector<c_Int32>
  &charList,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;

ccRect encloseCellRect( const cmStd vector<cmStd
  vector<c_Int32> >&charLineList,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;

ccRect encloseCellRect(const ccSynFontCharRenderParams&
  charParams = ccSynFontCharRenderParams()) const;
```

-
```
ccRect encloseCellRect(const cmStd vector<c_Int32>
  &charList,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;
```

Gets the enclosing cell rectangle of multiple characters.

Parameters

charList The list of characters.

charParams The character rendering parameters.

- ```
  ccRect encloseCellRect( const cmStd vector<cmStd
    vector<c_Int32> >&charLineList,
    const ccSynFontCharRenderParams& charParams =
    ccSynFontCharRenderParams()) const;
  ```

  Gets the enclosing cell rectangle of multiple characters (in a form convenient for multiple lines).

  Parameters

  charLineList The list of characters.

  charParams The character rendering parameters.

- ```
  ccRect encloseCellRect(const ccSynFontCharRenderParams&
    charParams = ccSynFontCharRenderParams()) const;
  ```

  Gets the enclosing cell rectangle of all characters.

  Parameters

  charParams The character rendering parameters.

markRect

```
ccRect markRect(c_Int32 charCode,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;
```

Gets the mark rectangle of one character.

Parameters

charCode The character code.

charParams The character rendering parameters.

encloseMarkRect

```
ccRect encloseMarkRect(const cmStd vector<c_Int32>
  &charList,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;

ccRect encloseMarkRect( const cmStd vector<cmStd
  vector<c_Int32> >&charLineList,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;

ccRect encloseMarkRect(const ccSynFontCharRenderParams&
  charParams = ccSynFontCharRenderParams()) const;
```

- ```
  ccRect encloseMarkRect(const cmStd vector<c_Int32>
    &charList,
    const ccSynFontCharRenderParams& charParams =
    ccSynFontCharRenderParams()) const;
  ```

  Gets the enclosing mark rectangle of multiple characters.

  Parameters

  charList The list of characters.

  charParams The character rendering parameters.

- ```
  ccRect encloseMarkRect( const cmStd vector<cmStd
    vector<c_Int32> >&charLineList,
    const ccSynFontCharRenderParams& charParams =
    ccSynFontCharRenderParams()) const;
  ```

  Gets the enclosing mark rectangle of multiple characters (in a form convenient for multiple lines).

  Parameters

  charLineList The list of characters.

  charParams The character rendering parameters.

- ```
  ccRect encloseMarkRect(const ccSynFontCharRenderParams&
    charParams = ccSynFontCharRenderParams()) const;
  ```

  Gets the enclosing mark rectangle of all characters.

  Parameters

  charParams The character rendering parameters.
  ```

renderMetrics

```
void renderMetrics(c_Int32 charCode,
  const ccSynFontRenderParams& params,
  c_UInt32 requestedMetrics,
  ccSynFontRenderMetrics& metrics) const;

void renderMetrics(const cmStd vector<c_Int32>& chars,
  const ccSynFontRenderParams& params,
  c_UInt32 requestedMetrics,
  ccSynFontRenderMetrics& metrics) const;

void renderMetrics(const ccCvlString& str,
  const ccSynFontRenderParams& params,
  c_UInt32 requestedMetrics,
  ccSynFontRenderMetrics& metrics) const;

void renderMetrics(const cmStd vector<cmStd vector<c_Int32>
  >& charLines,
  const ccSynFontRenderParams& params,
  c_UInt32 requestedMetrics,
  ccSynFontRenderMetrics& metrics) const;

void renderMetrics(const cmStd vector<ccCvlString>&
  strLines,
  const ccSynFontRenderParams& params,
  c_UInt32 requestedMetrics,
  ccSynFontRenderMetrics& metrics) const;
```

- ```
  void renderMetrics(c_Int32 charCode,
    const ccSynFontRenderParams& params,
    c_UInt32 requestedMetrics,
    ccSynFontRenderMetrics& metrics) const;
  ```

  Gets the metrics for a single character as the characters are rendered.

  Parameters

  charCode The code of the character.

  params Font rendering parameters.

  requestedMetrics The requested metrics.

  metrics The metrics.

  Throws

  ccSynFontDefs::UnknownChar Any character has **isKnown()** false.
  ```

- ```
  void renderMetrics(const cmStd vector<c_Int32>& chars,
    const ccSynFontRenderParams& params,
    c_UInt32 requestedMetrics,
    ccSynFontRenderMetrics& metrics) const;
  ```

  Gets the metrics for a single line of characters as the characters are rendered.

  Parameters

  chars The characters.

  params Font rendering parameters.

  requestedMetrics
  The requested metrics.

  metrics The metrics.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  void renderMetrics(const ccCvlString& str,
    const ccSynFontRenderParams& params,
    c_UInt32 requestedMetrics,
    ccSynFontRenderMetrics& metrics) const;
  ```

  Gets the metrics for a single line of characters represented as a string as the characters are rendered.

  Parameters

  str The string.

  params Font rendering parameters.

  requestedMetrics
  The requested metrics.

  metrics The metrics.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  void renderMetrics(const cmStd vector<cmStd vector<c_Int32>
    >& charLines,
    const ccSynFontRenderParams& params,
    c_UInt32 requestedMetrics,
    ccSynFontRenderMetrics& metrics) const;
  ```

  Gets the metrics for multiple lines of characters represented as a string as the characters are rendered.

  Parameters

  charLines The string of multiple lines of characters.

  params Font rendering parameters.

  requestedMetrics The requested metrics.

  metrics The metrics.

  Throws

  ccSynFontDefs::UnknownChar Any character has **isKnown()** false.

- ```
  void renderMetrics(const cmStd vector<ccCvlString>&
    strLines,
    const ccSynFontRenderParams& params,
    c_UInt32 requestedMetrics,
    ccSynFontRenderMetrics& metrics) const;
  ```

  Gets the metrics for multiple lines of characters represented as strings as the characters are rendered.

  Parameters

  strLines The strings of multiple lines of characters.

  params Font rendering parameters.

  requestedMetrics The requested metrics.

  metrics The metrics.

  Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

No rendering is actually performed. This function is typically used to determine the bounding rectangle of the string and to learn nominal positions of characters for use in training OCR and OCV.

Metrics vectors are ordered first by line and then by position, that is, all the metrics for the first line come first, followed by all the metrics for the second line, and so on.

renderRect

```
ccPelRect renderRect( c_Int32 charCode,
  const ccSynFontRenderParams& params,
  const cc2XformLinear& clientFromImage = cc2XformLinear())
  const;

ccPelRect renderRect( const cmStd vector<c_Int32>& chars,
  const ccSynFontRenderParams& params,
  const cc2XformLinear& clientFromImage = cc2XformLinear())
  const;

ccPelRect renderRect( const ccCvlString &str,
  const ccSynFontRenderParams& params,
  const cc2XformLinear& clientFromImage = cc2XformLinear())
  const;

ccPelRect renderRect( const cmStd vector<cmStd
  vector<c_Int32> >& charLines,
  const ccSynFontRenderParams& params,
  const cc2XformLinear& clientFromImage = cc2XformLinear())
  const;

ccPelRect renderRect( const cmStd vector<ccCvlString>&
  strLines,
  const ccSynFontRenderParams& params,
  const cc2XformLinear& clientFromImage = cc2XformLinear())
  const;
```

- ```
  ccPelRect renderRect( c_Int32 charCode,
    const ccSynFontRenderParams& params,
    const cc2XformLinear& clientFromImage = cc2XformLinear())
    const;
  ```

  Returns for a single character the rect that encloses all of the cell and mark rects of the character, padded as specified by *params*.The returned rect can then be used to pre-allocate an image to be used for rendering.
  ```

Parameters

charCode The code of the character.

params The parameters.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  ccPelRect renderRect( const cmStd vector<c_Int32>& chars,
    const ccSynFontRenderParams& params,
    const cc2XformLinear& clientFromImage = cc2XformLinear())
    const;
  ```

  Returns for a single line of characters the rect that encloses all of the cell and mark rects of the characters, padded as specified by *params*.The returned rect can then be used to pre-allocate an image to be used for rendering.

  Parameters

  chars The characters.

  params The parameters.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  ccPelRect renderRect( const ccCvlString &str,
    const ccSynFontRenderParams& params,
    const cc2XformLinear& clientFromImage = cc2XformLinear())
    const;
  ```

  Returns for a single line of characters represented as a string the rect that encloses all of the cell and mark rects of the characters, padded as specified by *params*.The returned rect can then be used to pre-allocate an image to be used for rendering.

  Parameters

  str The characters.

  params The parameters.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  ccPelRect renderRect( const cmStd vector<cmStd
    vector<c_Int32> >& charLines,
    const ccSynFontRenderParams& params,
    const cc2XformLinear& clientFromImage = cc2XformLinear())
    const;
  ```

  Returns for multiple lines of characters the rect that encloses all of the cell and mark rects of the characters, padded as specified by *params*.The returned rect can then be used to pre-allocate an image to be used for rendering.

  Parameters

  charLines The characters.

  params The parameters.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  ccPelRect renderRect( const cmStd vector<cccCvlString>&
    strLines,
    const ccSynFontRenderParams& params,
    const cc2XformLinear& clientFromImage = cc2XformLinear())
    const;
  ```

  Returns for multiple lines of characters represented as strings the rect that encloses all of the cell and mark rects of the characters, padded as specified by *params*.The returned rect can then be used to pre-allocate an image to be used for rendering.

  Parameters

  strLines The characters.

  params The parameters.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

render

```
void render(ccPelBuffer<c_UInt8>& image,
  c_Int32 charCode,
  const ccSynFontRenderParams& params) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cmStd vector<c_Int32>& chars,
  const ccSynFontRenderParams& params) const;

void render(ccPelBuffer<c_UInt8>& image,
  const ccCvlString& str,
  const ccSynFontRenderParams& params) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cmStd vector<cmStd vector<c_Int32> >& chars,
  const ccSynFontRenderParams& params) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cmStd vector<ccCvlString>& str,
  const ccSynFontRenderParams& params) const;

void render(ccPelBuffer<c_UInt8>& image,
  c_Int32 charCode,
  const ccSynFontRenderParams& params,
  ccSynFontRenderResult& result) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cmStd vector<c_Int32>& chars,
  const ccSynFontRenderParams& params,
  ccSynFontRenderResult& result) const;

void render(ccPelBuffer<c_UInt8>& image,
  const ccCvlString& str,
  const ccSynFontRenderParams& params,
  ccSynFontRenderResult& result) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cmStd vector<cmStd vector<c_Int32> >& chars,
  const ccSynFontRenderParams& params,
  ccSynFontRenderResult& result) const;

void render(ccPelBuffer<c_UInt8>& image,
  const cmStd vector<ccCvlString>& str,
  const ccSynFontRenderParams& params,
  ccSynFontRenderResult& result) const;
```

-
```
void render(ccPelBuffer<c_UInt8>& image,
  c_Int32 charCode,
  const ccSynFontRenderParams& params) const;
```

  Renders a character into pelbuffer using the specified parameters.

If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

Parameters

image The image.

charCode The code of the character.

params The parameters.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const cmStd vector<c_Int32>& chars,
    const ccSynFontRenderParams& params) const;
  ```

Renders a single line of characters into pelbuffer using the specified parameters.

If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

Parameters

image The image.

chars The characters.

params The parameters.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const ccCvlString& str,
    const ccSynFontRenderParams& params) const;
  ```

Renders a single line of characters represented as a string into pelbuffer using the specified parameters.

If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

Parameters

image The image.

str The characters.

params The parameters.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const cmStd vector<cmStd vector<c_Int32> >& chars,
    const ccSynFontRenderParams& params) const;
  ```

  Renders multiple lines of characters into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

  image The image.

  chars The characters.

  params The parameters.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const cmStd vector<ccCvlString>& str,
    const ccSynFontRenderParams& params) const;
  ```

  Renders multiple lines of characters represented as strings into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

  image The image.

  str The characters.

params The parameters.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    c_Int32 charCode,
    const ccSynFontRenderParams& params,
    ccSynFontRenderResult& result) const;
  ```

  Renders a single character into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

  image The image.

  charCode The code of the character.

  params The parameters.

  result The result.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const cmStd vector<c_Int32>& chars,
    const ccSynFontRenderParams& params,
    ccSynFontRenderResult& result) const;
  ```

  Renders a single line of characters into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

  image The image.

  chars The characters.

  params The parameters.

  result The result.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const ccCvlString& str,
    const ccSynFontRenderParams& params,
    ccSynFontRenderResult& result) const;
  ```

  Renders a single line of characters represented as a string into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

  image The image.

  str The characters.

  params The parameters.

  result The result.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const cmStd vector<cmStd vector<c_Int32> >& chars,
    const ccSynFontRenderParams& params,
    ccSynFontRenderResult& result) const;
  ```

  Renders multiple lines of characters into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

  image The image.

  chars The characters.

  params The parameters.

  result The result.

Throws

ccSynFontDefs::UnknownChar
Any character has **isKnown()** false.

- ```
  void render(ccPelBuffer<c_UInt8>& image,
    const cmStd vector<ccCvlString>& str,
    const ccSynFontRenderParams& params,
    ccSynFontRenderResult& result) const;
  ```

  Renders multiple lines of characters represented as strings into pelbuffer using the specified parameters.

  If the image is unbound, it will be allocated using **renderRect()** with identity client coordinates. If the image is bound, it should initially have been set to the background color by the user.

  Parameters

image	The image.
str	The characters.
params	The parameters.
result	The result.

  Throws

  ccSynFontDefs::UnknownChar
  Any character has **isKnown()** false.

availableChars

```
const cmStd vector<c_Int32>& availableChars(bool
  includeAllBlanks = false) const;
```

Gets a list of characters available in the font.

If *includeAllBlanks* is true and the font has multiple blank characters, then all the blanks will be included; if *includeAllBlanks* is false, then only the **defaultBlank()** blank character will be included.

The list includes all printable characters plus the space character (if present).

Parameters

includeAllBlanks
```

availableNonblankChars

```
const cmStd vector<c_Int32>& availableNonblankChars()
  const;
```

Gets a list of characters available in the font. The list includes all printable characters.

availableBlankChars

```
const cmStd vector<c_Int32>& availableBlankChars() const;
```

Gets a list of blank (that is, space) characters available in the font.

hasBlank

```
bool hasBlank() const;
```

Returns whether this font contains at least one blank character.

defaultBlank

```
c_Int32 defaultBlank() const;
```

Returns the default blank character for this font.

Throws

ccSynFontDefs::UnknownChar
The **hasBlank()** operation returned false.

advance

```
cc2Vect advance(c_Int32 charCode,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;
```

Returns the advance for the specified character.

Parameters

charCode The code of the character.

charParams The character rendering parameters.

Notes

The total advance from one character to the next:

advance(a) + **kerning(a, b)** + **tracking()**

renderMetrics() is often a more convenient way to get this information for a particular string.

Throws

ccSynFontDefs::UnknownChar
The **isKnown(***charCode***)** operation returned false.

kerning

```
cc2Vect kerning(c_Int32 firstCharCode,
  c_Int32 secondCharCode,
  const ccSynFontCharRenderParams& charParams =
  ccSynFontCharRenderParams()) const;
```

Returns the kerning between the specified ordered pair of characters.

Parameters

firstCharCode The code of the first character.

secondCharCode
The code of the second character.

charParams The character rendering parameters.

Notes

The total advance from one character to the next:

advance(a) + **kerning(a, b)** + **tracking()**

renderMetrics() is often a more convenient way to get this information for a particular string.

Throws

ccSynFontDefs::UnknownChar
The **isKnown(***charCode***)** operation returned false.

isProportional

```
bool isProportional() const;
```

Returns whether this font is a proportional font.

fontType

```
FontType fontType() const;
```

Gets the type of the font as one of the following types:

eNone, *eRaster*, *eStroke*, *eOutline*, or *eImage*

isImageFont

```
bool isImageFont() const;
```

Returns whether the font is an image font; that is, whether it is based on stored bitmaps of characters rather than on synthetic descriptions of characters.

Notes

A font whose *fontType* is *eImage* will always be an image font, but it is also possible (confusingly) to have a font whose type is *eOutline* be an image font.

canMoveImageFontCharOrigins

```
bool canMoveImageFontCharOrigins() const;
```

Returns whether the font is an image font whose characters can have their origins moved using **moveImageFontCharOrigins()**.

moveImageFontCharOrigins

```
bool moveImageFontCharOrigins( ccImageFontChar::Origin
  origin = ccImageFontChar::eDetectedMarkLowerLeft);
```

Moves the origins as specified if the font is an image font.

Parameters

origin The origin.

Returns true if the move request could be fulfilled; that is, if the font was an image font.

Notes

Only fonts whose font type is *eImage* can be modified this way; the *eOutline* fonts that are image fonts cannot be modified like this (and typically do not need to be).

fontNames

```
static void fontNames(const ccCvlString& filename,
  cmStd vector<ccCvlString>& fontNames);
```

Gets names of all fonts present in the font file.

Parameters

filename The name of the font file.

fontNames The names of the fonts present in the font file.

Deprecated Members

The following member functions are deprecated. Use the overloads that take a **ccSynFontCharRenderParams** instead.

markRect

```
ccRect markRect(c_Int32 charCode,
  double spotSizeFactor,
  double extraStrokeWidth = 0.,
  double spotSpacingXScale = 1.0,
  double spotSpacingYScale = 1.0) const;
```

Gets the mark rectangle of one character.

Parameters

charCode The code of the character.

spotSizeFactor The *spotSize* factor.

extraStrokeWidth The increase of the stroke width of the font.

spotSpacingXScale
The x-coordinate spot spacing factor.

spotSpacingYScale
The y-coordinate spot spacing factor.

encloseMarkRect

```
ccRect encloseMarkRect(const cmStd vector<c_Int32>
  &charList,
  double spotSizeFactor,
  double extraStrokeWidth = 0.,
  double spotSpacingXScale = 1.0,
  double spotSpacingYScale = 1.0) const;

ccRect encloseMarkRect( const cmStd vector<cmStd
  vector<c_Int32> >&charLineList,
  double spotSizeFactor,
  double extraStrokeWidth = 0.,
  double spotSpacingXScale = 1.0,
  double spotSpacingYScale = 1.0) const;

ccRect encloseMarkRect(double spotSizeFactor,
  double extraStrokeWidth = 0.,
  double spotSpacingXScale = 1.0,
  double spotSpacingYScale = 1.0) const;
```

-
```
ccRect encloseMarkRect(const cmStd vector<c_Int32>
  &charList,
  double spotSizeFactor,
  double extraStrokeWidth = 0.,
  double spotSpacingXScale = 1.0,
  double spotSpacingYScale = 1.0) const;
```

Gets the enclosing mark rectangle of multiple characters.

Parameters

charList The characters.

spotSizeFactor The *spotSize* factor.

extraStrokeWidth The increase of the stroke width of the font.

spotSpacingXScale
The x-coordinate spot spacing factor.

spotSpacingYScale
The y-coordinate spot spacing factor.

- ```
  ccRect encloseMarkRect( const cmStd vector<cmStd
    vector<c_Int32> >&charLineList,
    double spotSizeFactor,
    double extraStrokeWidth = 0.,
    double spotSpacingXScale = 1.0,
    double spotSpacingYScale = 1.0) const;
  ```

  Gets the enclosing mark rectangle of multiple characters (in a form convenient for multiple lines).

  Parameters

  charLineList The characters.

  spotSizeFactor The *spotSize* factor.

  extraStrokeWidth The increase of the stroke width of the font.

  spotSpacingXScale
  The x-coordinate spot spacing factor.

  spotSpacingYScale
  The y-coordinate spot spacing factor.

- ```
  ccRect encloseMarkRect(double spotSizeFactor,
    double extraStrokeWidth = 0.,
    double spotSpacingXScale = 1.0,
    double spotSpacingYScale = 1.0) const;
  ```

  Get the enclosing mark rectangle of all characters.

  Parameters

  spotSizeFactor The *spotSize* factor.

  extraStrokeWidth The increase of the stroke width of the font.

  spotSpacingXScale
  The x-coordinate spot spacing factor.

  spotSpacingYScale
  The y-coordinate spot spacing factor.

ccSynFont

ccSynFontCharRenderParams

```
#include <ch_cvl/synfont.h>

class ccSynFontCharRenderParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccSynFontCharRenderParams** class specifies parameters that control the rendering of individual characters.

Constructors/Destructors

ccSynFontCharRenderParams

```
explicit ccSynFontCharRenderParams(double spotSizeFactor =
  1.0,
  double extraStrokeWidth = 0.,
  double spotSpacingXScale = 1.0,
  double spotSpacingYScale = 1.0):
    spotSizeFactor_(spotSizeFactor),
    extraStrokeWidth_(extraStrokeWidth),
    spotSpacingXScale_(spotSpacingXScale),
    spotSpacingYScale_(spotSpacingYScale);
```

Contstructs a default character render parameters object.

Notes

The default copy constructor, assign operator, and destructor are used.

Parameters

spotSizeFactor The *spotSize* factor.

extraStrokeWidth The increase of the stroke width of the font.

spotSpacingXScale
The x-coordinate spot spacing factor.

spotSpacingYScale
The y-coordinate spot spacing factor.

spotSizeFactor The *spotSize* factor.

extraStrokeWidth The increase of the stroke width of the font.

spotSpacingXScale
The x-coordinate spot spacing factor.

spotSpacingYScale
The y-coordinate spot spacing factor.

Public Member Functions

spotSizeFactor

```
double spotSizeFactor() const;

void spotSizeFactor(double s);
```

- `double spotSizeFactor() const;`

Gets a factor to be used as a multiplier on the font's *spotSize* value.

- `void spotSizeFactor(double s);`

Sets a factor to be used as a multiplier on the font's *spotSize* value.

Parameters
s The new *spotSize* factor value.

spotSize determines the size of the spots for raster fonts and the width of the stroke for stroke fonts.

Has no effect on outline fonts.

Throws
ccSynFontDefs::BadParams
spotSizeFactor is negative.

extraStrokeWidth

```
double extraStrokeWidth() const;

void extraStrokeWidth(double e);
```

- `double extraStrokeWidth() const;`

Gets the amount by which to change the stroke width of the given font, using morphology.

- `void extraStrokeWidth(double e);`

  Sets the amount by which to change the stroke width of the given font, using morphology.

  Parameters

  e The increase of the stroke width of the font.

  Using **spotSizeFactor()** is usually preferable, but this function may be needed for outline fonts.

  Notes

  The *extraStrokeWidth* value may be negative to indicate stroke width should be decreased.

spotSpacingXScale

```
double spotSpacingXScale() const;
void spotSpacingXScale(double xScale);
```

- `double spotSpacingXScale() const;`

  Gets the x-coordinate spot spacing factor.

- `void spotSpacingXScale(double xScale);`

  Sets the x-coordinate spot spacing factor.

  Parameters

  xScale The new x-coordinate spot spacing factor.

  spotSpacingXScale() specifies a factor by which to change the x-coordinate of spots relative to each other; also changes the endpoints of strokes.

  Throws

  ccSynFontDefs::BadParams

  xScale is less than or equal to 0.

spotSpacingYScale

```
double spotSpacingYScale() const;

void spotSpacingYScale(double yScale);
```

- `double spotSpacingYScale() const;`

  Gets the y-coordinate spot spacing factor.

- `void spotSpacingYScale(double yScale);`

  Sets the y-coordinate spot spacing factor.

  spotSpacingYScale() specifies a factor by which to change the y-coordinate of spots relative to each other; also changes the endpoints of strokes.

  Parameters

  yScale The new y-coordinate spot spacing factor.

  Throws

  ccSynFontDefs::BadParams

  yScale is less than or equal to 0.

ccSynFontDefs

```
#include <ch_cvl/synfont.h>

class ccSynFontDefs;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

The **ccSynFontDefs** class contains all of the enumerations specific to the ccSynFont class and its supporting classes.

Enumerations

Polarity

```
enum Polarity;
```

Polarity types for characters to specify the appearance of the character relative to its nearby background in the image.

Value	Meaning
eDarkOnLight = 0	Dark characters on light background
eLightOnDark	Light characters on dark background
eUnknown	Unknown polarity

ccSynFontDefs

ccSynFontRenderMetrics

```
#include <ch_cvl/synfont.h>

class ccSynFontRenderMetrics;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccSynFontRenderMetrics** class contains metrics (such as bounding boxes and origins) resulting from rendering one or more characters.

Constructors/Destructors

ccSynFontRenderMetrics

```
ccSynFontRenderMetrics();
```

Default constructor, defaults to all false.

Notes

The default copy constructor, assign operator, and destructor are used.

Enumerations

Metrics

```
enum Metrics;
```

Types of metrics for rendered characters.

Value	Meaning
ePose = 0x01	Character pose
eCellRect = 0x02	Character cell rect
eMarkRect = 0x04	Character mark rect
eUnionRect = 0x08	Encloses cell rect and mark rect
eOrigin = 0x10	Character origin

Value	Meaning
eEncloseCellRect = 0x20	Character enclose cell rect
ePolarity = 0x40	Character polarity

Public Member Functions

reset

```
void reset();
```

Resets all metrics to being not computed (all values to false).

hasPoses

```
bool hasPoses() const;
void hasPoses(bool h);
```

- `bool hasPoses() const;`

  Gets whether or not the object has a valid poses value.

- `void hasPoses(bool h);`

  Sets whether or not the object has a valid poses value.

  Parameters

  h — If true, the object has a valid poses value.

poses

```
const cmStd vector<cc2Xform>& poses() const;
void poses(const cmStd vector<cc2Xform>& poses);
```

- `const cmStd vector<cc2Xform>& poses() const;`

  Gets the vector of character poses.

  Throws

  ccSynFontDefs::NotComputed
  The **hasPoses()** operation returned false.

- `void poses(const cmStd vector<cc2Xform>& poses);`

  Sets the vector of character poses.

Parameters
poses The new vector of character poses.

hasCellRects

```
bool hasCellRects() const;

void hasCellRects(bool h);
```

- `bool hasCellRects() const;`

  Gets whether or not the object has a valid *cellRects* value.

- `void hasCellRects(bool h);`

  Sets whether or not the object has a valid *cellRects* value.

  Parameters
  h If true, the object has a valid *cellRects* value.

cellRects

```
const cmStd vector<ccAffineRectangle>& cellRects() const;

void cellRects(const cmStd vector<ccAffineRectangle>&
  cellRects);
```

- `const cmStd vector<ccAffineRectangle>& cellRects() const;`

  Gets the vector of character cellrects.

  Throws
  ccSynFontDefs::NotComputed
  The **hasCellRects()** operation returned false.

- `void cellRects(const cmStd vector<ccAffineRectangle>& cellRects);`

  Sets the vector of character cellrects.

  Parameters
  cellRects The new vector of character cellrects.

hasMarkRects

```
bool hasMarkRects() const;

void hasMarkRects(bool h);
```

- `bool hasMarkRects() const;`

  Gets whether or not the object has a valid *markRects* value.

- `void hasMarkRects(bool h);`

  Sets whether or not the object has a valid *markRects* value.

  Parameters

  h If true, the object has a valid *markRects* value.

markRects

```
const cmStd vector<ccAffineRectangle>& markRects() const;

void markRects(const cmStd vector<ccAffineRectangle>&
 markRects);
```

- `const cmStd vector<ccAffineRectangle>& markRects() const;`

  Gets the vector of character markrects.

  Throws

  ccSynFontDefs::NotComputed
  The **hasMarkRects()** operation returned false.

- `void markRects(const cmStd vector<ccAffineRectangle>& markRects);`

  Sets the vector of character markrects.

  Parameters

  markRects The new vector of character markrects.

hasUnionRects

```
bool hasUnionRects() const;

void hasUnionRects(bool h);
```

- `bool hasUnionRects() const;`

  Gets whether or not the object has a valid *unionRects* value.

- `void hasUnionRects(bool h);`

  Sets whether or not the object has a valid *unionRects* value.

  Parameters

  h If true, the object has a valid *unionRects* value.

unionRects

```
const cmStd vector<ccAffineRectangle>& unionRects() const;

void unionRects(const cmStd vector<ccAffineRectangle>&
unionRects);
```

- `const cmStd vector<ccAffineRectangle>& unionRects() const;`

  Gets the vector of character unionrects.

  Throws

  ccSynFontDefs::NotComputed
  The **hasUnionRects()** operation returned false.

- `void unionRects(const cmStd vector<ccAffineRectangle>& unionRects);`

  Sets the vector of character unionrects.

  Parameters

  unionRects The new vector of character unionrects.

hasOrigins

```
bool hasOrigins() const;

void hasOrigins(bool h);
```

- `bool hasOrigins() const;`

  Gets whether or not the object has a valid origins value.

- `void hasOrigins(bool h);`

  Sets whether or not the object has a valid origins value.

  Parameters

  h If true, the object has a valid origins value.

origins

```
const cmStd vector<cc2Vect>& origins() const;
void origins(const cmStd vector<cc2Vect>& origins);
```

- `const cmStd vector<cc2Vect>& origins() const;`

  Gets the vector of character origins.

  Throws
  ccSynFontDefs::NotComputed
  The **hasOrigins()** operation returned false.

- `void origins(const cmStd vector<cc2Vect>& origins);`

  Sets the vector of character origins.

  Parameters
  origins The new vector of character origins.

hasEncloseCellRects

```
bool hasEncloseCellRects() const;
void hasEncloseCellRects(bool h);
```

- `bool hasEncloseCellRects() const;`

  Gets whether or not the object has a valid *encloseCellRects* value.

- `void hasEncloseCellRects(bool h);`

  Sets whether or not the object has a valid *encloseCellRects* value.

  Parameters
  h If true, the object has a valid *encloseCellRects* value.

encloseCellRects

```
const cmStd vector<ccRect>& encloseCellRects() const;

void encloseCellRects(const cmStd vector<ccRect>&
encloseCellRects);
```

- `const cmStd vector<ccRect>& encloseCellRects() const;`

  Gets the vector of character enclosing cellrects.

  Throws

  ccSynFontDefs::NotComputed

  The **hasEncloseCellRects()** operation returned false.

- `void encloseCellRects(const cmStd vector<ccRect>& encloseCellRects);`

  Sets the vector of character enclosing cellrects.

  Parameters

  encloseCellRects

  The new vector of character enclosing cellrects.

hasPolarities

```
bool hasPolarities() const;

void hasPolarities(bool h);
```

- `bool hasPolarities() const;`

  Gets whether or not the object has a valid polarities value.

- `void hasPolarities(bool h);`

  Sets whether or not the object has a valid polarities value.

  Parameters

  h If true, the object has a valid polarities value.

polarities

```
const cmStd vector<ccSynFontDefs::Polarity>& polarities()
  const;

void polarities(const cmStd
  vector<ccSynFontDefs::Polarity>& p);
```

- ```
  const cmStd vector<ccSynFontDefs::Polarity>& polarities()
    const;
  ```

  Gets the vector of polarities.

- ```
  void polarities(const cmStd
    vector<ccSynFontDefs::Polarity>& p);
  ```

  Sets the vector of polarities.

  Parameters

p	The new vector of polarities.

ccSynFontRenderOutline

```
#include <ch_cvl/synfont.h>

class ccSynFontRenderOutline;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccSynFontRenderOutline** class represents a character or set of characters as a shape and as a graphic object. The outline has a hierarchical structure, with the top level of the structure corresponding to individual characters.

Constructors/Destructors

ccSynFontRenderOutline

```
ccSynFontRenderOutline();

ccSynFontRenderOutline(const ccSynFontRenderOutline& rhs);
```

- `ccSynFontRenderOutline();`

  Constructs a default (empty) outline.

- `ccSynFontRenderOutline(const ccSynFontRenderOutline& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

Operators

operator=

```
ccSynFontRenderOutline& operator=(const
  ccSynFontRenderOutline& rhs);
```

Assignment operator.

Parameters

rhs The assignment source.

Notes

The default destructor is used.

Public Member Functions

reset

```
void reset();
```

Resets the outline to a default (empty) state.

hasShape

```
bool hasShape() const;
```

Gets whether the outline has a representation as a shape.

shape

```
ccShapePtrh_const shape() const;
```

Gets the outline represented as a shape.

Throws

ccSynFontDefs::NotComputed
The **hasShape()** operation returned false.

hasGraphics

```
bool hasGraphics() const;
```

Gets whether the outline has a representation as a graphic object.

graphics

```
ccGraphicPtrh_const graphics() const;
```

Gets the outline represented as a graphic object.

Throws

ccSynFontDefs::NotComputed
The **hasGraphics()** operation returned false.

draw

```
void draw(ccGraphicList& glist) const;
```

Draws the outline into the graphic list.

Note There is no effect if **hasGraphics()** is false.

Parameters

glist The graphic list.

numChildren

```
c_Int32 numChildren() const;
```

Returns the number of children.

children

```
const cmStd vector<ccSynFontRenderOutline>& children()
  const;

void children(const cmStd vector<ccSynFontRenderOutline>&
  children);
```

- ```
  const cmStd vector<ccSynFontRenderOutline>& children()
    const;
  ```

  Gets the children of this outline.

- ```
  void children(const cmStd vector<ccSynFontRenderOutline>&
    children);
  ```

  Sets the children of this outline, and updates **shape()** and **graphics()** to represent the combination of all of them.

  Parameters

  children The new children of this outline.

takeChildren

```
void takeChildren(cmStd vector<ccSynFontRenderOutline>&
  children);
```

Sets the children of this outline, and updates **shape()** and **graphics()** to represent the combination of all of them; takes ownership of the vector, which will be empty after calling this function.

Parameters

children The new children of this outline.

isNull

```
bool isNull() const;
```

Returns whether this outline is null; for example, will not draw anything.

Notes

The outline for a blank character will always be null.

ccSynFontRenderOutline

props

```
const ccGraphicProps& props() const;
void props(const ccGraphicProps& p);
```

- `const ccGraphicProps& props() const;`

  Gets the graphic properties.

- `void props(const ccGraphicProps& p);`

  Sets the graphic properties.

  Parameters

  p The new graphic properties.

  Notes

  It is possible for an outline to have multiple properties; the getter will return the properties of the first non-null child, if there is one.

color

```
ccColor color() const;
void color(const ccColor& color);
```

- `ccColor color() const;`

  Gets the color.

- `void color(const ccColor& color);`

  Sets the color.

  Parameters

  color The new color.

  Notes

  It is possible for an outline to have multiple colors; the getter will return the color of the first non-null child, if there is one.

ccSynFontRenderParams

```
#include <ch_cvl/synfont.h>

class ccSynFontRenderParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccSynFontRenderParams** class specifies parameters that control the rendering of one or more characters as well as specifying which results are requested by a rendering operation.

Constructors/Destructors

ccSynFontRenderParams

```
ccSynFontRenderParams(): tracking_(cc2Vect(0, 0)),
  foreground_(10),
  background_(200),
  padding_(0),
  preserveImageCharacterGraylevels_(false),
  requestedResults_(eImage),
  requestedMetrics_(0),
  doSimplifyOutline_(false),
  outlineProps_(ccColor::greenColor());
```

Constructs a default font render parameters object.

Notes

The default copy constructor, assignment operator, and destructor are used.

Enumerations

Results

```
enum Results;
```

Value	Meaning
eImage = 0x0100	Produces the rendered image.
eMetrics = 0x0200	Produces the metrics specified by **requestedMetrics()**
eRenderedRects = 0x0400	Produces the rects for the characters.
eRenderedMasks = 0x0800	Produces the masks for the characters.
eOutline = 0x1000	Produces an outline for the set of characters.

Public Member Functions

tracking

```
cc2Vect tracking() const;
void tracking(const cc2Vect& t);
```

- `cc2Vect tracking() const;`

  Gets the tracking value, the distance between the origins of adjacent characters on a line.

- `void tracking(const cc2Vect& t);`

  Sets the tracking value, the distance between the origins of adjacent characters on a line.

  Parameters

  t The new tracking value.

extraLeading

```
cc2Vect extraLeading() const;
void extraLeading(const cc2Vect& e);
```

- `cc2Vect extraLeading() const;`

  Gets the amount to add to the font's leading value, the distance between the origins of sequential lines.

- `void extraLeading(const cc2Vect& e);`

  Sets the amount to add to the font's leading value, the distance between the origins of sequential lines.

  Parameters

  e The new amount to add to the font's leading value.

charParams

```
const ccSynFontCharRenderParams& charParams() const;
void charParams(const ccSynFontCharRenderParams& c);
```

- `const ccSynFontCharRenderParams& charParams() const;`

  Gets all character rendering parameters as a set.

- `void charParams(const ccSynFontCharRenderParams& c);`

  Sets all character rendering parameters as a set.

  Parameters

  c The new character rendering parameters as a set.

spotSizeFactor

```
double spotSizeFactor() const;
void spotSizeFactor(double s);
```

- `double spotSizeFactor() const;`

  Gets a factor to be used as a multiplier on the font's *spotSize* value.

- `void spotSizeFactor(double s);`

  Sets a factor to be used as a multiplier on the font's *spotSize* value.

  Parameters

  s The new *spotSize* factor value.

  spotSize determines the size of the spots for raster fonts and the width of the stroke for stroke fonts.

  Has no effect on outline fonts.

  Throws

  ccSynFontDefs::BadParams
  spotSizeFactor is negative.

extraStrokeWidth

```
double extraStrokeWidth() const;
void extraStrokeWidth(double e);
```

- `double extraStrokeWidth() const;`

  Gets the increase of the stroke width of the font.

- `void extraStrokeWidth(double e);`

  Sets the increase of the stroke width of the font.

  Parameters

  e The increase of the stroke width of the font.

  extraStrokeWidth() increases the stroke width of the given font, using morphology.

  Using **spotSizeFactor()** is usually preferable, but this function may be needed for outline fonts.

  Notes

  The *extraStrokeWidth* value may be negative to indicate stroke width should be decreased.

spotSpacingXScale

```
double spotSpacingXScale() const;
void spotSpacingXScale(double xScale);
```

- `double spotSpacingXScale() const;`

  Gets the x-coordinate spot spacing factor.

- `void spotSpacingXScale(double xScale);`

  Sets the x-coordinate spot spacing factor.

  Parameters
  xScale The new x-coordinate spot spacing factor.

  spotSpacingXScale() specifies a factor by which to change the x-coordinate of spots relative to each other; also changes the endpoints of strokes.

  Throws
  BadParams
  xScale is less than or equal to 0.

spotSpacingYScale

```
double spotSpacingYScale() const;
void spotSpacingYScale(double yScale);
```

- `double spotSpacingYScale() const;`

  Gets the y-coordinate spot spacing factor.

- `void spotSpacingYScale(double yScale);`

  Sets the y-coordinate spot spacing factor.

  spotSpacingYScale() specifies a factor by which to change the y-coordinate of spots relative to each other; also changes the endpoints of strokes.

  Parameters
  yScale The new y-coordinate spot spacing factor.

  Throws
  BadParams
  yScale is less than or equal to 0.

clientFromFontXform

```
cc2Xform clientFromFontXform() const;
void clientFromFontXform(const cc2Xform& x);
```

- `cc2Xform clientFromFontXform() const;`

  Gets the client-from-font xform to be used for rendering.

- `void clientFromFontXform(const cc2Xform& x);`

  Sets the client-from-font xform to be used for rendering.

  Parameters

  x The new client-from-font xform to be used for rendering.

foreground

```
c_UInt8 foreground() const;
void foreground(c_UInt8 f);
```

- `c_UInt8 foreground() const;`

  Gets the foreground color to be used for rendering.

- `void foreground(c_UInt8 f);`

  Sets the foreground color to be used for rendering.

  Parameters

  f The new foreground color to be used for rendering.

background

```
c_UInt8 background() const;
void background(c_UInt8 b);
```

- `c_UInt8 background() const;`

  Gets the background color to be used for rendering.

- `void background(c_UInt8 b);`

  Sets the background color to be used for rendering.

Parameters

b The new background color to be used for rendering.

preserveImageCharacterGraylevels

```
bool preserveImageCharacterGraylevels() const;

void preserveImageCharacterGraylevels(bool p);
```

- `bool preserveImageCharacterGraylevels() const;`

  Gets whether to preserve the graylevels if the font is an image font, instead of trying to map the graylevels to the requested foreground/background range.

- `void preserveImageCharacterGraylevels(bool p);`

  Sets whether to preserve the graylevels if the font is an image font, instead of trying to map the graylevels to the requested foreground/background range.

  Parameters

  p If true, graylevels are preserved if the font is an image font.

  However, the foreground/background values will still indicate whether the image should be inverted.

padding

```
c_Int32 padding() const;

void padding(c_Int32 p);
```

- `c_Int32 padding() const;`

  Gets the uniform padding amount to surround the rendering.

- `void padding(c_Int32 p);`

  Sets the uniform padding amount to surround the rendering.

  Parameters

  p The new uniform padding amount to surround the rendering.

requestedResults

```
c_UInt32 requestedResults() const;
void requestedResults(c_UInt32 r);
```

- `c_UInt32 requestedResults() const;`

  Gets the types of results that the rendering operation should produce as a bitwise-OR of values from **ccSynFontRenderParams::Results**.

- `void requestedResults(c_UInt32 r);`

  Sets the types of results that the rendering operation should produce as a bitwise-OR of values from **ccSynFontRenderParams::Results**.

  The following types are possible:

  eImage – produces the rendered image.

  eMetrics – produces the metrics specified by **requestedMetrics()**

  eRenderedRects – produces the rects for the characters.

  eRenderedMasks – produces the masks for the characters.

  eOutline – produces an outline for the set of characters.

  The specified types of results will be stored in a **ccSynFontRenderResult** object.

  Parameters

r	The result type.

requestedMetrics

```
c_UInt32 requestedMetrics() const;
void requestedMetrics(c_UInt32 r);
```

- `c_UInt32 requestedMetrics() const;`

  Gets the types of metrics that the rendering operation should produce as a bitwise-OR of values from **ccSynFontRenderMetrics::Metrics**.

- `void requestedMetrics(c_UInt32 r);`

  Sets the types of metrics that the rendering operation should produce as a bitwise-OR of values from **ccSynFontRenderMetrics::Metrics**.

  Notes

  The **requestedResults()** must also specify *eMetrics* to actually compute any of the requested metrics. Each metric corresponds to the similarly-named accessor in a **ccSynFontRenderMetrics** object.

  Parameters

r	The types of metrics.

doSimplifyOutline

```
bool doSimplifyOutline() const;
void doSimplifyOutline(bool s);
```

- `bool doSimplifyOutline() const;`

  Gets whether to simplify the outline if one is requested.

- `void doSimplifyOutline(bool s);`

  Sets whether to simplify the outline if one is requested.

  If false, then the outlines will be the results of edge detection; if it is true, then simpler, smoother contours will be produced.

  Parameters

s	If true, the outline is simplified if one is requested.

outlineProps

```
const ccGraphicProps& outlineProps() const;
void outlineProps(const ccGraphicProps& p);
```

- `const ccGraphicProps& outlineProps() const;`

  Gets the graphic properties for the outline.

- `void outlineProps(const ccGraphicProps& p);`

  Sets the graphic properties for the outline.

Note The color is part of the outline properties, so changes to the outline color (see **outlineColor()** below) will change the color stored in the properties.

Parameters

p The graphic properties for the outline.

outlineColor

```
ccColor outlineColor() const;
void outlineColor(const ccColor& c);
```

- `ccColor outlineColor() const;`

  Gets the color for the outline.

- `void outlineColor(const ccColor& c);`

  Sets the color for the outline.

  Note The color is part of the outline properties, so changes to the properties (See **outlineProps()** above) will also affect the outline color.

  Parameters

  c The color for the outline.

ccSynFontRenderResult

```
#include <ch_cvl/synfont.h>

class ccSynFontRenderResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

The **ccSynFontRenderResult** class contains the result of a rendering operation, which can include the rendered image, an outline, and/or various metrics.

Constructors/Destructors

ccSynFontRenderResult

```
ccSynFontRenderResult();

ccSynFontRenderResult(const ccSynFontRenderResult& rhs);
```

- `ccSynFontRenderResult();`

  Constructs a default (empty) render result.

- `ccSynFontRenderResult(const ccSynFontRenderResult& rhs);`

  Copy constructor.

  Parameters

  rhs The source of the copy.

  Notes

  The default destructor is used.

Operators

operator=

```
ccSynFontRenderResult& operator=(const
  ccSynFontRenderResult& rhs);
```

Assignment operator.

Parameters

rhs The assignment source.

Public Member Functions

reset

```
void reset();
```

Resets result to be empty.

hasImage

```
bool hasImage() const;
```

Gets whether the result contains an image.

image

```
ccPelBuffer_const<c_UInt8> image() const;
```

Gets the rendered image.

Notes

For efficiency, the image stored in the result is a shallow copy of the image passed to **render()**, so any changes the user may make to the pels of that image will also be made to the pels of the image stored in this result.

Throws

ccSynFontDefs::NotComputed
The **hasImage()** operation returned false.

hasMetrics

```
bool hasMetrics() const;
```

Gets whether the result contains metrics.

metrics

```
const ccSynFontRenderMetrics& metrics() const;
```

Gets the rendered metrics.

Throws

ccSynFontDefs::NotComputed
The **hasMetrics()** operation returned false.

hasRenderedRects

```
bool hasRenderedRects() const;
```

Gets whether the result contains rendered character rectangles.

renderedRects

```
const cmStd vector<ccPelRect>& renderedRects() const;
```

Gets the rendered character rectangles. Each rectangle specifies the image-coordinate rectangle of pixels that enclose the rendered pixels for one character.

Throws

ccSynFontDefs::NotComputed
The **hasRenderedRects()** operation returned false.

hasRenderedMasks

```
bool hasRenderedMasks() const;
```

Gets whether the result contains rendered masks.

renderedMasks

```
const cmStd vector<ccPelBuffer_const<c_UInt8> >&
  renderedMasks() const;
```

Gets the rendered character masks. Each mask is an image whose window is the corresponding rendered rect and which contains a 255 for any pixel touched by rendering this character and a zero otherwise.

Throws

ccSynFontDefs::NotComputed
The **hasRenderedMasks()** operation returned false.

hasOutline

```
bool hasOutline() const;
```

Gets whether the result contains an outline.

outline

```
const ccSynFontRenderOutline& outline() const;
```

Gets the outline.

Throws

ccSynFontDefs::NotComputed
The **hasOutline()** operation returned false.

ccSynFontRenderResult

ccThreadID

```
#include <ch_cvl/threads.h>

class ccThreadID: public cc_PtrHandleBase_RepAccessor;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	No

A platform-independent thread handle. You can obtain a **ccThreadID** when you create a thread by calling **cfCreateThread()**, or by when you call **cfGetCurrentThreadID()**.

Constructors/Destructors

ccThreadID

```
ccThreadID();
```

You should not construct this class directly.

Operators

operator=

```
ccThreadID& operator=(const ccThreadID& rhs);
```

Assigns one **ccThreadID** to another.

Parameters

rhs The **ccThreadID** from which to assign.

operator const void*

```
operator const void* () const;
```

Returns zero if this is a default constructed **ccThreadID**, a nonzero value otherwise.

operator!

```
bool operator!() const;
```

Returns true if this is a non-default constructed **ccThreadID**, false otherwise.

operator==

```
bool operator==(const ccThreadID& rhs) const;
```

Returns true if this **ccThreadID** is equivalent to the supplied **ccThreadID**.

Parameters

rhs The **ccThreadID** to compare to this one.

operator!=

```
bool operator!=(const ccThreadID& rhs) const;
```

Returns false if this **ccThreadID** is equivalent to the supplied **ccThreadID**.

Parameters

rhs The **ccThreadID** to compare to this one.

Public Member Functions

osDependentThreadId

```
c_UInt32 osDependentThreadId() const;
```

Return the underlying operating system's thread identifier.

osDependentThreadHandle

```
void *osDependentThreadHandle() const;
```

Return the underlying operating system's thread representation.

atexit

```
void atexit(void (*func)
  (const ccThreadID&, void *arg), void *arg) const;
```

Registers a function which will be called after this thread exits. The supplied function must have the following signature:

```
void function(const ccThreadID& thr, void *arg);
```

where *thr* is the thread calling the function and *arg* is the argument you supply when you call **atexit()**.

You cannot call this function if this **ccThreadID** refers to the main program thread; use the C runtime function **atexit()** instead.

Parameters

func The function to call on exit

arg A value that will be supplied as the second argument to *func* when it is called.

Throws

ccThreadID::CouldNotRegister

The function could not be registered. Make sure that this **ccThreadID** does not refer to the main program thread and that you have not attempted to register more than 16 functions.

Notes

This function will be run in the context of the calling thread.

Static Functions

mainThreadID

```
static ccThreadID mainThreadID();
```

Returns a **ccThreadID** representing this program's main thread.

ccThreadID

ccThreadLocal

```
#include <ch_cvl/threads.h>

template <class T>class ccThreadLocal: public cc_ThreadLocal;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A templated class that implements thread-safe global storage.

You can instantiate any number of **ccThreadLocal** objects as static or global variables. Each thread that later accesses the contents of a **ccThreadLocal** object gets its own instance of the object. The **ccThreadLocal** class includes functions that let you set and get the value of the object.

You specify the type for a **ccThreadLocal** object on construction, as shown below:

```
ccThreadLocal<ccIPair> resultLocation;
```

The class you specify must support copy construction.

Constructors/Destructors

ccThreadLocal `ccThreadLocal();`

Constructs a **ccThreadLocal** object of a given type. You should only construct **ccThreadLocal** objects at STI time. You should not create automatic **ccThreadLocal** objects or dynamically allocated **ccThreadLocal** objects.

Public Member Functions

value

```
T* value();

const T* value() const;

void value(const T& val);
```

- `T* value();`

  Returns a pointer to the object stored in this **ccThreadLocal**. If the value of this **ccThreadLocal** has not been set, this function returns 0.

- `const T* value() const;`

  Returns a **const** pointer to the object stored in this **ccThreadLocal**. If the value of this **ccThreadLocal** has not been set, this function returns 0.

- `void value(const T& val);`

  Sets the value of this **ccThreadLocal**. This function is thread-safe: each thread sets its own value for a given **ccThreadLocal** object.

  Parameters

val	A reference to the object to store in this **ccThreadLocal**. The object being referred to is copied into the **ccThreadLocal** object.

ccThresholdResult

```
#include <ch_cvl/thresh.h>

class ccThresholdResult;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class represents the results of running the Threshold tool.

Constructors/Destructors

ccThresholdResult

```
ccThresholdResult();
```

Constructs a threshold result object with no computed threshold or score.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns true if the threshold was successfully computed; false otherwise.

threshold

```
c_Int32 threshold() const;
```

Returns the computed threshold.

Throws

ccThresholdDefs::NotComputed
The threshold was not computed.

score

```
double score() const;
```

Returns the score for the threshold computation. This score is a value in the range 0.0 through 1.0 that indicates the separation of the two groups of bins determined by the threshold. A greater score means more separation. The score is 0.0 if the partial histogram has only one nonzero bin.

Throws

ccThresholdDefs::NotComputed
The threshold was not computed.

ccTimeout

```
#include <ch_cvl/attent.h>

class ccTimeout : public ccAttentionClient ;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

The **ccTimeout** class implements a timeout facility for CVL Vision Tools. You use **ccTimeout** by constructing a **ccTimeout** with a specified value (in seconds), then calling a vision tool function. If the tool has not completed its operation within the specified period of time, the tool throws a *ccTimeout::Expired* exception.

The **ccTimeout**-based timeout mechanism is not supported by all CVL vision tools. Refer to the documentation for specific tools to determine if that tool supports **ccTimeout**-based timeouts.

Constructors/Destructors

ccTimeout

```
ccTimeout(double delay_s);
```

Constructs this **ccTimeout** object to wait the specified number of seconds, and then throw *ccTimeout::Expired*. The error is thrown from any CVL function that supports timeouts

Once this **ccTimeout** object expires and throws *ccTimeout::Expired*, this **ccTimeout** object is deactivated.

If this **ccTimeout** object is destroyed before its time elapses, no error is thrown.

Parameters

delay_s — The timeout value seconds. *delay_s* must be greater than or equal to 0.0.

Notes

The timeout is expressed in real elapsed time, not execution time of the tool. The timeout you specify is rounded down to the nearest value supported by **ccTimer**, based on **ccTimer** granularity.

ccTimeout

ccTimeoutProp

```
#include <ch_cvl/prop.h>

class ccTimeoutProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes the timeout property of an acquisition FIFO queue. The timeout property limits the amount of time an acquisition FIFO will wait to obtain any resources it needs to perform an acquisition. For example, if all available image buffers are in use, the acquisition FIFO will wait until the end of the timeout period for a buffer to become available. If a buffer does not become available within that time, the acquisition FIFO gives up and the acquisition fails. The failure is reported in a **ccAcqFailure** object: **ccAcqFailure::isTimeout()** returns true.

In manual mode and semi-automatic mode the timeout interval begins when you call **start()**. Note that if you queue several starts, their timeout timers run coincident with one another. For auto trigger mode acquisitions, the timeout interval begins when the previous acquisition completes, and queued request timeout timers run serially, in the order in which they were queued.

For an externally triggered acquisition, the acquisition FIFO will wait indefinitely for a hardware trigger once it has obtained all the necessary resources.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Note This class has been deprecated and is maintained for backward compatibility only. New code should use other methods to protect against extended blocking during image acquisition.

Constructors/Destructors

ccTimeoutProp

```
ccTimeoutProp();
explicit ccTimeoutProp(double);
```

- `ccTimeoutProp();`

  Creates a new timeout property not associated with any FIFO. The timeout period is set to *ccExposureProp::defaultTimeout*.

- `explicit ccTimeoutProp(double seconds);`

  Creates a new timeout property not associated with any FIFO. The timeout period is set to *seconds*.

  Parameters

  seconds — The timeout period in seconds. A value of zero means to time out if the acquisition cannot start immediately. A value of *HUGE_VAL* means that the acquisition never times out.

  Throws

  ccTimeoutProp::BadParams — *seconds* was less than zero.

Public Member Functions

timeout

```
void timeout(double seconds);
double timeout();
```

- `void timeout(double seconds);`

  Set the timeout duration to *seconds*.

  Parameters

  seconds — The timeout period in seconds. A value of zero means to time out if the acquisition cannot start immediately. A value of *HUGE_VAL* means that the acquisition never times out.

  Throws

  ccTimeoutProp::BadParams — *seconds* was less than zero.

- `double timeout();`

  Return the timeout period in seconds.

Constants

defaultTimeout `static const double defaultTimeout = HUGE_VAL;`

ccTimeoutProp

ccTimer

```
#include <ch_cvl/timer.h>

class ccTimer;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

A class that implements a simple stopwatch-style timer. You can use this class to measure how long operations take in real time.

Constructors/Destructors

ccTimer

```
ccTimer(bool running = false);
```

Constructs a **ccTimer**. The **ccTimer** is initialized to have an elapsed time of 0 and an iteration count of 1.

Parameters

running — If true, the **ccTimer** begins accumulating elapsed time when it is constructed. If false, the **ccTimer** needs to be started with a call to **start()**.

Public Member Functions

start

```
void start();
```

Starts accumulating elapsed time. Calling this function does not reset this **ccTimer**'s total elapsed time.

stop

```
void stop();
```

Stops accumulating elapsed time. Calling this function does not reset this **ccTimer**'s total elapsed time.

Notes

You can obtain the current elapsed time for a **ccTimer** while it is running.

running

```
bool running() const;
```

Returns true if this **ccTimer** is currently accumulating elapsed time, false if it is stopped.

reset

```
void reset();
```

Resets this **ccTimer**'s accumulated elapsed time to 0.

Notes

You do not need to stop a **ccTimer** to reset it; you can reset a running **ccTimer**.

count

```
c_Int32 count() const;
void count(c_Int32 loop);
```

- `c_Int32 count() const;`

Returns this **ccTimer**'s iteration count. The **ticks()**, **sec()**, **msec()**, and **usec()** functions all normalize their return values by dividing them by the iteration count.

- `void count(c_Int32 loop);`

Sets this **ccTimer**'s iteration count. The **ticks()**, **sec()**, **msec()**, and **usec()** functions all normalize their return values by dividing them by the iteration count.

On construction, a **ccTimer**'s iteration count is 1.

Parameters

loop	The iteration count. *loop* must be greater than 0.

ticks

```
double ticks() const;
```

Returns the accumulated elapsed time in ticks, divided by this **ccTimer**'s iteration count.

sec

```
double sec() const;
```

Returns the accumulated elapsed time in seconds, divided by this **ccTimer**'s iteration count.

msec

```
double msec() const;
```

Returns the accumulated elapsed time in milliseconds, divided by this **ccTimer**'s iteration count.

usec

```
double usec() const;
```

Returns the accumulated elapsed time in microseconds, divided by this **ccTimer**'s iteration count.

rawTicks

```
double rawTicks() const;
```

Reports this **ccTimer**'s current accumulated elapsed time in ticks. The value returned by this function is not normalized by the iteration count, and is not adjusted to correct for the overhead of starting and stopping the timer.

Call the **ticksPerSecond()** to determine the duration of a tick.

Notes

Raw ticks are represented internally using 53 bits. The raw tick count wraps each 2^{53} ticks.

Static Functions

resolution

```
static double resolution();
```

Returns this system's underlying timer resolution, in microseconds per raw tick.

sleep

```
static void sleep(double seconds);
```

Calling this function causes the current thread to block for the requested number of seconds, rounded up to the underlying timer facility's tick granularity.

Parameters

seconds	The number of seconds to block. *seconds* must be greater than or equal to 0.

Notes

If you specify a value of 0.0, the current thread relinquishes the remainder of its timeslice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, this function returns immediately, and the thread continues execution.

While **sleep()** is accurate to approximately 10 milliseconds, **ccTimer** itself is accurate to within a few microseconds.

ticksPerSecond

```
static c_UInt32 ticksPerSecond();
```

Returns this system's underlying timer resolution, in raw ticks per second.

readSystemTime `static double readSystemTime();`

Returns the current system time in raw ticks.

Macros

cmExecutionTime

`cmExecutionTime(tm, statement);`

Executes the supplied C++ expression one time and places the time required to execute it in the supplied **ccTimer**.

Parameters

tm	A **ccTimer**.
statement	A C++ expression.

cmTiming `cmTiming(tm, num, statement);`

Executes the supplied C++ expression the specified number of times and places the time required to execute it in the supplied **ccTimer**.

Parameters

tm	A **ccTimer**.
num	The number of times to execute *statement*.
statement	A C++ expression.

Notes

This time returned by this macro includes the overhead of executing a **for** loop.

cmAutoTime `cmAutoTime(tm, min_ticks, statement);`

Executes the supplied C++ expression until the specified number of timer ticks has elapsed.

Parameters

tm	A **ccTimer**.
min_ticks	The number of timer ticks.
statement	A C++ expression.

ccTriggerFilterProp

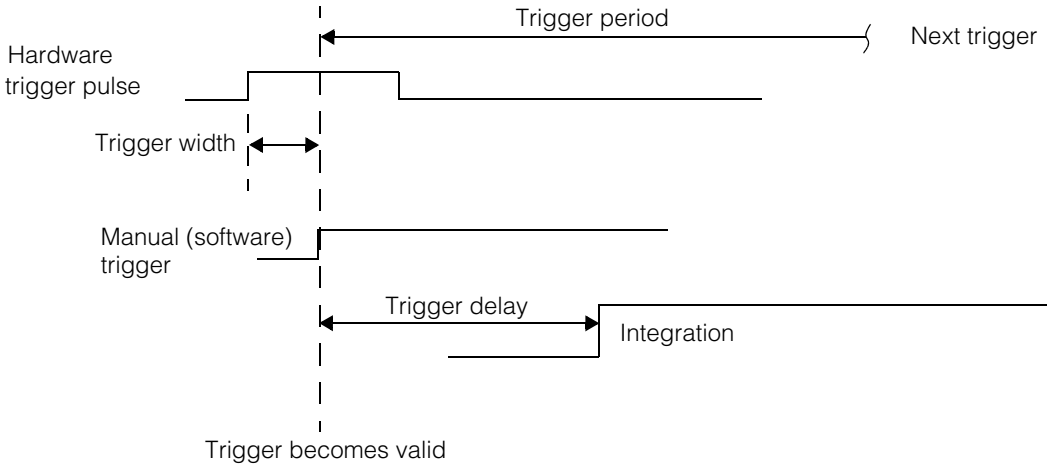
```
#include <ch_cvl/prop.h>

class ccTriggerFilterProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This object encapsulates the input trigger properties; trigger width, trigger period, and trigger delay. See the following diagram.

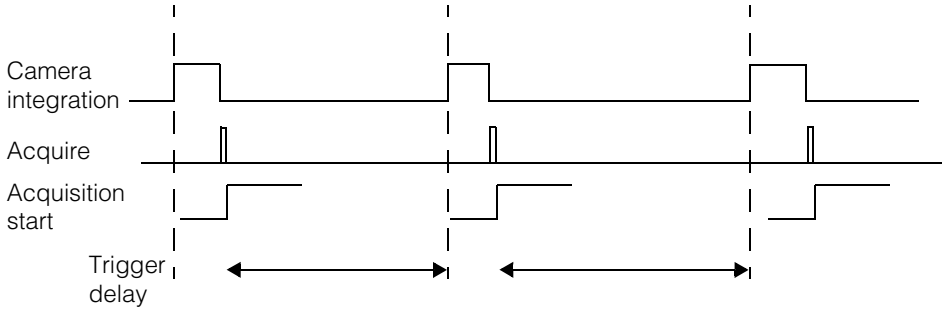


The example above shows both a hardware trigger and a manual trigger which is initiated by software. A hardware trigger pulse must satisfy the width requirement to become valid, while a manual trigger is effective immediately.

If you specify a trigger delay, camera integration is delayed by this amount after the trigger is effective.

For platforms such as the MVS-8600 and MVS-8600e that do not support setting variable steps-per-line for a test encoder, the trigger delay can be used with manual triggers to set the time between line scans for line scan camera applications. Without a trigger

delay the application would run at the camera frame rate, which may be faster than you wish. You can set a trigger delay to slow down the camera to the desired line rate. See the following diagram.



Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccTriggerFilterProp

```
ccTriggerFilterProp();

ccTriggerFilterProp(const ccTriggerFilterProp& that);

ccTriggerFilterProp(
  double width,
  double period,
  double delay);

virtual ~ccTriggerFilterProp() {};
```

- `ccTriggerFilterProp();`

  Default constructor.

- `ccTriggerFilterProp(const ccTriggerFilterProp& that);`

  Copy constructor.

Parameters

that	The object to copy.

- ```
  ccTriggerFilterProp(
    double width,
    double period,
    double delay);
  ```

  Constructs an object with the specified trigger width, period, and delay.

  Parameters

width	The hardware trigger width requirement in seconds. The trigger input signal must be asserted for at least this duration before it is recognized as a valid input trigger. If the trigger does not meet this width constraint, it is ignored.
period	The trigger input period requirement, in seconds. Only the first valid trigger within a *period* will initiate a camera integration cycle. Other valid triggers in that same period result in **isMissed()** errors. This can be used to help limit the camera acquisition rate. A zero specifies there is no period requirement.
delay	The delay in seconds, from an accepted trigger to camera integration start.

- ```
  virtual ~ccTriggerFilterProp() {};
  ```

  Destructor.

Operators

operator==

```
bool operator==(const ccTriggerFilterProp& that) const;
```

Comparison operator. Returns true is this object = *that*. Returns false otherwise.

Parameters

that	The other **ccTriggerFilterProp** object.

operator!=

```
bool operator!=(const ccTriggerFilterProp& that) const;
```

Comparison operator. Returns false is this object = *that*. Returns true otherwise.

Parameters

that The other **ccTriggerFilterProp** object.

Public Member Functions

triggerWidth

```
void triggerWidth(double);
double triggerWidth() const;
```

Note This method is not supported any longer.

The hardware trigger width requirement in seconds. The trigger input signal must be asserted for at least this duration before it is recognized as a valid input trigger. If the trigger does not meet this width constraint, it is ignored.

Notes

The latency between the edge transition of the trigger signal and the start of camera integration is computed as (triggerWidth + triggerDelay).

- `void triggerWidth(double);`

  Sets a new trigger width.

  Parameters

  double The new trigger width in seconds.

- `double triggerWidth() const;`

  Returns the current trigger width.

triggerPeriod

```
void triggerPeriod(double);
double triggerPeriod() const;
```

Note This method is not supported any longer.

The trigger input period requirement, in seconds. Only the first valid trigger within a *period* will initiate a camera integration cycle. Other valid triggers in that same period result in **isMissed(**) errors.

This can be used to help limit the camera acquisition rate. A zero specifies there is no period requirement.

- `void triggerPeriod(double);`

  Sets a new trigger period.

  Parameters

  double The new trigger period.

- `double triggerPeriod() const;`

  Returns the current trigger period.

triggerDelay

```
void triggerDelay(double delay);
double triggerDelay() const;
```

The delay in seconds, from an accepted trigger to camera integration start.

- `void triggerDelay(double);`

  Sets a new trigger delay.

  Parameters

  delay The new trigger delay in seconds.

  Notes

  On MVS-8600 and MVS-8600e frame grabbers using a linescan camera, the trigger delay property takes effect for every line instead of every frame as with area scan cameras. When using this property with linescan cameras it has a resolution of one horizontal line.

  For example, setting the delay to 1.0 adds a delay of one horizontal line time between acquisition of successive lines.

  This property may be used in conjunction with the built-in test encoder on the MVS-8600 to adjust the rate of acquisition of successive lines (line rate).

- `double triggerDelay() const;`

  Returns the current trigger delay.

ignoreMissedTrigger

```
void ignoreMissedTrigger(bool);

bool ignoreMissedTrigger() const;
```

- `void ignoreMissedTrigger(bool);`

  Returns true if a missed trigger does not generate an **isMissed** failure, false if a missed trigger does generate the failure.

  Parameters
  bool

- `bool ignoreMissedTrigger() const;`

  Set this value to false to cause a missed trigger to generate an **isMissed** failure when using an area scan camera. A missed trigger is a hardware trigger which failed to cause an acquisition to occur.

  See **ccAcqFailure** on page 209for conditions under which this failure may be generated.

  The default is true, which means that a missed trigger will not generate an **isMissed** failure.

  Notes
  Only the Rev 4 and above MVS-8602e supports this function. Rev 2 MVS-8602e does not support this function.

Constants

defaultTriggerWidth

```
static const double defaultTriggerWidth;
```

Default = 5e-6.

defaultTriggerPeriod

```
static const double defaultTriggerPeriod;
```

Default = 0.

defaultTriggerDelay

```
static const double defaultTriggerDelay;
```

Default = 0.

ccTriggerFilterProp

ccTriggerModel

```
#include <ch_cvl/trigmodl.h>

class ccTriggerModel;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This class describes the trigger models available for acquisition FIFO queues. The acquisition software creates trigger model objects that you can use as parameters to the **ccTriggerProp::triggerModel()** method. The following table summarizes the trigger models:

Trigger type	Also known as	Acquisition started by	Associated global function
Manual trigger	Software trigger	**ccAcqFifo::start()**	**cfManualTrigger()**
Auto trigger	Hardware trigger	External signal	**cfAutoTrigger()**
Semi trigger	n/a	**ccAcqFifo::start()**, then external signal	**cfSemiTrigger()**
Slave trigger	n/a	Same as master	**cfSlaveTrigger()**
Free run trigger	n/a	**ccAcqFifo:triggerEnable()** set to true	**cfFreeRunTrigger()**

If you are using a vision board that supports CVM11, you can specify encoder triggered acquisition. This is a special case of hardware (auto) triggering when using line scan cameras with CVM11. To use encoder triggering, set your trigger model as Auto or Semi, then use **ccEncoderProp::encoderTriggerEnabled()** as described on page 1363.

The relationship between the non-slave trigger models is summarized as follows:

	Needs external trigger	Does not need external trigger
Needs start()	Semi	Manual (software)
Does not need start()	Auto (hardware)	Free Run

Constructors/Destructors

The acquisition software creates one object for each trigger model.

Enumerations

ceBufferImages `enum ceBufferImages;`

This enumeration controls how the FIFO buffers completed images.

Value	Meaning
ckBuffer_Compatible	Images are buffered to the buffer capacity of the FIFO. This setting is appropriate for applications that must process all images. This is the default setting for all built-in trigger models.
ckBufferOnlyLatest	New images replace older images in the FIFO. This setting is appropriate for live video or applications that do not need to process every acquired image. Do not use this setting if **ccTriggerModel::startAction()** indicates that **start()** must be called, or with synchronous configurations. Replaced images are converted to **isMissed** errors, which are then handled according to the **missedErrorHandling()** setting. For example, if **ckIgnoreMissed** is set, replaced images are simply discarded. Take care when using callbacks with **ckBufferOnlyLatest** and **ckIgnoreMissed**. Because images are discarded, you will not be able to call **ccAcqFifo::complete()** as many times as your complete callback routine is called.

ceTriggerSource `enum ceTriggerSource;`

This enumeration specifies when the camera should acquire an image.

Value	Meaning
ckHardwareTrigger	Acquire an image when the hardware detects a trigger.
ckImmediateTrigger	Acquire an image as soon as the hardware is ready. This setting together with **ccTriggerModel::startAction(ckNotAllowed_Compatible)** causes the FIFO to "free run" or acquire images as fast as the acquisition hardware and camera will allow.
ckSoftwareTrigger	Wait until **ccAcqFifo::trigger()** is called before acquiring an image. This setting allows more precise control over when the image is captured than would be possible with **ccAcqFifo::start()**.

ceMissedErrorHandling
`enum ceMissedErrorHandling;`

This enumeration specifies whether the FIFO should report **isMissed** errors.

Value	Meaning
ckReportMissed_Compatible	Report all errors in a backward-compatible manner. This is the default setting for all built-in trigger models.
ckIgnoreMissed	Suppress reporting of **isMissed** errors; report all other errors as usual. This setting is useful when the video system is intentionally being triggered too fast.

ceStartAction `enum ceStartAction;`

This enumeration specifies whether acquisitions must be explicitly requested using **ccAcqFifo()::start()**, and determines how the start routine responds to error conditions.

Value	Meaning
ckRequestAcquire_Compatible	**ccAcqFifo::start()** must be called for every acquisition. This setting is the default for the manual and semi trigger models. If an acquisition cannot be requested because of insufficient resources, an **isMissed** error is queued and returned in sequence using **ccAcqFifo::complete()**. This setting is appropriate when property settings can be changed between acquisitions, or when multiple FIFOs are sharing a single camera.
ckNotAllowed_Compatible	Acquisitions can be started without a call to **ccAcqFifo::start()**. Calling this method will in fact cause a **ccAcqFifo::StartNotAllowed** exception to be thrown. This setting is the default for the auto and slave trigger models. This setting allows the FIFO to begin preparing the hardware for the next acquisition immediately upon completion of the preceding acquisition. This setting is appropriate for high-performance applications that use the same acquisition properties for every acquisition.
ckRequestAcquireOrFail	Similar to *ckRequestAcquire_Compatible*, but if a request fails, **ccAcqFifo::start()** returns false instead of queueing an **isMissed** error.
ckNotAllowedSilent	Similar to *ckNotAllowed_Compatible*, but calls to **ccAcqFifo::start()** are ignored instead of generating an exception.

ceHardwareTriggerAction

```
enum ceHardwareTriggerAction;
```

This enumeration specifies the choices you have for controlling image acquisition in response to a hardware trigger.

Value	Meaning
ckStartAcq_Compatible	Causes the acquisition fifo to acquire one image when an hardware trigger is input. Only one image is acquired for one hardware trigger. This is the default setting for Cognex hardware when using the built-in auto trigger model.
ckStartStopAcq	Causes the acquisition fifo to start acquiring images when a hardware trigger is input. The fifo will continue to acquire images as fast as possible until another hardware trigger is received at which point the fifo will stop acquiring. The fifo will wait for the next hardware trigger to occur to resume acquiring. Multiple images can be acquired from a single hardware trigger.
ckAcqOnTriggerHigh	Causes the acquisition fifo to start acquiring images and to continue acquiring images as long as the hardware trigger input signal is held high. The sense of the trigger input high setting depends upon the *triggerLowToHigh* property. Multiple images can be acquired from a single hardware trigger.
ckRestartAcq	This trigger model should be used with line scan cameras only. An initial trigger causes acquisition to begin at the start of the next image. Successive image acquisitions occur until the next trigger. The trigger can come any time, from immediately, to many images down stream. Each successive trigger causes acquisition to restart at the beginning of the next image. If a trigger occurs during an acquisition, the acquisition is terminated immediately and the image acquired up to that point is returned. Acquisition then restarts at the beginning of the next image.

Public Member Functions

name

```
const TCHAR* name() const;
```

Returns the name of this trigger model as a string. For example, “Manual” or “Automatic”.

copyForCustomization

```
ccTriggerModel* copyForCustomization() const;
```

Creates a user modifiable trigger model initialized with the settings of this trigger model. As all built-in CVL trigger models are **const**, you must call this method to customize any trigger model settings.

Notes

The returned trigger model is dynamically allocated using **new()**, therefore you must delete it when no longer needed to prevent memory leaks. A convenient way to do this is to use the reference-counted pointer handle version, **ccTriggerModelPtrh**. For example:

```
const ccTriggerModel tm1 = cfManualTrigger();
ccTriggerModelPtrh tm2Ptr(
  tm1.copyForCustomization());
// Now you can customize the copy pointed to by tm2Ptr
```

bufferImages

```
ceBufferImages bufferImages() const;

void bufferImages(ceBufferImages buffer);
```

- `ceBufferImages bufferImages() const;`

Retrieves the image buffer setting.

- `void bufferImages(ceBufferImages buffer);`

Sets how the FIFO buffers images.

Parameters

buffer The image buffer setting. Must be one of:

ccTriggerModel::ckBuffer_Compatible
ccTriggerModel::ckBufferOnlyLatest

Throws

ccTriggerModel::BadParams
buffer is not one of the allowed values.

triggerSource

```
ceTriggerSource triggerSource() const;
void triggerSource(ceTriggerSource source);
```

- `ceTriggerSource triggerSource() const;`

  Retrieves the trigger source setting.

- `void triggerSource(ceTriggerSource source);`

  Sets the trigger source to specify when the camera acquires an image.

  Parameters

  source The trigger source. Must be one of:

  ccTriggerModel::ckHardwareTrigger
  ccTriggerModel::ckImmediateTrigger
  ccTriggerModel::ckSoftwareTrigger

  Throws

  ccTriggerModel::BadParams
  source is not one of the allowed values.

missedErrorHandling

```
ceMissedErrorHandling missedErrorHandling() const;
void missedErrorHandling(ceMissedErrorHandling how);
```

- `ceMissedErrorHandling missedErrorHandling() const;`

  Retrieves the missed error setting, which determines how the FIFO reports **isMissed()** errors.

- `void missedErrorHandling(ceMissedErrorHandling how);`

  Sets how the FIFO reports **isMissed()** errors.

  Parameters

  how The missed error reporting setting. Must be one of:

  ccTriggerModel::ckReportMissed_Compatible
  ccTriggerModel::ckIgnoreMissed

Throws

ccTriggerModel::BadParams

how is not one of the allowed values.

hardwareTriggerAction

```
ceHardwareTriggerAction hardwareTriggerAction() const;

void hardwareTriggerAction(
                              ceHardwareTriggerAction action);
```

This setting controls how an external hardware trigger will affect acquisition. The choices are explained in the **ceHardwareTriggerAction** enum on page 3149.

- `ceHardwareTriggerAction hardwareTriggerAction() const;`

  Returns the current hardware trigger action setting.

- ```
  void hardwareTriggerAction(
                                ceHardwareTriggerAction action);
  ```

  Sets a new hardware trigger action. Must be one of the **ceHardwareTriggerAction** enums.

  Parameters

  action The new hardware trigger action.

  Notes

  hardwareTriggerAction() is only valid when you are using a hardware trigger, and **startAction()** is *ckNotAllowed_Compatible*.

  Only MVS-8600 frame grabbers operating with line scan cameras support this hardware trigger action functionality.

startAction

```
ceStartAction startAction() const;

void startAction(ceStartAction start);
```

- `ceStartAction startAction() const;`

  Retrieves the start action setting, which determines whether **ccAcqFifo::start()** must be called to start an acquisition.
```

- `void startAction(ceStartAction start);`

  Sets whether **ccAcqFifo::start()** must be called to start an acquisition. This setting also determines the response of the start routine to error conditions.

  Parameters

  start The start action setting. Must be one of:

  ccTriggerModel::ckRequestAcquire_Compatible
  ccTriggerModel::ckNotAllowed_Compatible
  ccTriggerModel::ckRequestAcquireOrFail
  ccTriggerModel::ckNotAllowedSilent

  Throws

  ccTriggerModel::BadParams
  start is not one of the allowed values.

bufferHardwareTrigger

```
bool bufferHardwareTrigger() const;

void bufferHardwareTrigger(bool enable);
```

- `bool bufferHardwareTrigger() const;`

  Retrieves the current enable/disable setting of the hardware trigger buffer feature.

- `void bufferHardwareTrigger(bool enable);`

  Controls whether triggers received during camera readout are accepted as a valid trigger without generating an **isMissed()** error. When enabled and such a trigger is received, the trigger is buffered until the acquisition is complete. At that time, the trigger is issued internally to initiate the next acquisition.

  Notes

  If multiple triggers occur during the readout, only the first is buffered. All other such triggers are ignored, and no **isMissed()** error is generated for these ignored triggers.

  If any triggers occur before the charge transfer interval, which precedes the camera readout phase, then an **isMissed()** error is generated.

  This feature is only supported on certain Cognex frame grabbers.

Deprecated Members

The following member functions are deprecated. They are provided for backward compatibility only.

isSlave

```
bool isSlave() const;
```

Returns whether this trigger model is slave.

usesInputLine

```
bool usesInputLine() const;
```

Returns whether this trigger model uses an input line to trigger acquisitions.

canStart

```
bool canStart() const;
```

Returns true if **ccAcqFifo::start()** can safely be called to start an acquisition with this trigger model. Returns false if calling **ccAcqFifo::start()** would throw an error.

Notes

This function returns true for **cfManualTrigger()** and **cfSemiTrigger()** and false for **cfAutoTrigger()** and **cfSlaveTrigger()**.

Typedefs

ccTriggerModelPtrh

```
typedef ccPtrHandle<ccTriggerModel> ccTriggerModelPtrh;
```

ccTriggerProp

```
#include <ch_cvl/prop.h>

class ccTriggerProp : virtual public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

This class describes the trigger property of an acquisition FIFO queue. Trigger models are described in **ccTriggerModel** on page 3145.

Since this class is one of the base classes of **ccAcqProps**, you do not need to create an instance of this class. All classes derived from **ccAcqFifo** can use **ccAcqFifo::properties()** to return a properties object that includes this property.

A table showing the acquisition hardware platforms that support this property is included in the *Acquiring Images* chapter of the *CVL User's Guide*.

Constructors/Destructors

ccTriggerProp

```
ccTriggerProp();

explicit ccTriggerProp(const ccTriggerModel& model,
  bool enable = true, bool lowToHigh = true,
  const ccPtrHandle<ccAcqFifo>& triggerMaster =
    ccPtrHandle<ccAcqFifo>());
```

- ```
  ccTriggerProp();
  ```

  Create a new trigger property object not associated with any FIFO. Triggering is enabled and default trigger model is **cfManualTrigger()**.

- ```
  explicit ccTriggerProp(const ccTriggerModel& model,
    bool enable = true, bool lowToHigh = true,
    const ccPtrHandle<ccAcqFifo>& triggerMaster =
      ccPtrHandle<ccAcqFifo>());
  ```

  Creates a new trigger not associated with any FIFO with the specified trigger model.

Parameters

model	The trigger model to use with this trigger property; one of **cfManualTrigger()**, **cfAutoTrigger()**, **cfSemiTrigger()** or **cfSlaveTrigger()**.
enable	True if triggered acquisitions are enabled. False if acquisitions should be immediate.
lowToHigh	True if a trigger signal is an input line transition from low to high. False means that a trigger signal is an input line transition from high to low
triggerMaster	If model is **cfSlaveTrigger()**, this is the slave's master acquisition FIFO.

Public Member Functions

triggerModel

```
void triggerModel(const ccTriggerModel& model);
const ccTriggerModel& triggerModel();
```

- `void triggerModel(const ccTriggerModel& model);`

  Set the trigger model for this trigger to *model*. The trigger model property determines the acquisition method. Changing the trigger model will temporarily disable triggers (if enabled) and flush the FIFO and all slaves.

  See **ccTriggerModel** on page 3145 for more information.

  Parameters

model	The trigger model to use with this trigger property; one of **cfManualTrigger()**, **cfAutoTrigger()**, **cfSemiTrigger()**, or **cfSlaveTrigger()**.

- `const ccTriggerModel& triggerModel();`

  Return this trigger property's trigger model.

triggerEnable

```
void triggerEnable(bool enable);
bool triggerEnable();
```

- `void triggerEnable(bool enable);`

  Enable or disable acquisitions as specified by the trigger model. The default is true. **triggerEnable()** affects all related master/slave FIFOs identically. For best results use the following recommendations:

  - Use **triggerEnable()** on a master only, if possible. In a multi-threaded master/slave application, undefined behavior can occur if two threads call **triggerEnable()** on separate, but related, FIFOs at the same time.
  - Disable triggers before changing trigger models, especially when setting up master/slave applications. This prevents extraneous disabling and enabling of triggers inside CVL functions during the master/slave setup.

  If an acquisition FIFO has slaves, changing the state of the master changes the states of all of the slaves.

  Parameters

  enabled — True if acquisitions can take place in all triggering models. False if acquisitions are to be queued (trigger model is **cfManualTrigger()**) or if transitions on the trigger input line are to be ignored (trigger model is **cfAutoTrigger()** or **cfSemiTrigger()**).

- `bool triggerEnable();`

  Return true if acquisitions are enabled for the specified trigger model.

triggerLowToHigh

```
void triggerLowToHigh(bool polarity);
bool triggerLowToHigh() const;
```

- `void triggerLowToHigh(bool polarity);`

  Sets the input trigger line polarity. The default is true: low to high transition.

  Parameters

  polarity — True means that a trigger signal is an input line transition from low to high. False means that a trigger signal is an input line transition from high to low.

Notes

Consult your hardware manual for proper connections and polarity.

- `bool triggerLowToHigh() const;`

Returns the input trigger line polarity.

triggerMaster

```
void triggerMaster(const ccPtrHandle<ccAcqFifo>& master);
const ccPtrHandle<ccAcqFifo>& triggerMaster() const;
```

- `void triggerMaster(const ccPtrHandle<ccAcqFifo>& master);`

Set the master of the acquisition FIFO to be *master*. This setting is relevant only when the trigger model is **cfSlaveTrigger()**.

Parameters

master The acquisition FIFO that is the master of this acquisition FIFO.

Throws

ccAcqFifo::InvalidMaster

This acquisition FIFO cannot be a slave to the specified master. See **couldSlaveTo()** on page 3158.

- `const ccPtrHandle<ccAcqFifo>& triggerMaster();`

Returns this acquisition FIFO's master.

couldSlaveTo

```
bool couldSlaveTo(const ccPtrHandle<ccAcqFifo>& master)
  const;
```

Returns true if this acquisition FIFO can be the slave of the specified acquisition FIFO. Returns false if these properties are not associated with a FIFO or if *master* is not a suitable master for this FIFO.

Parameters

master The acquisition FIFO to test.

Notes

ccTriggerProp::couldSlaveTo() indicates whether one FIFO can be a slave to another without taking into account current property settings. In particular, the master port property of the FIFOs must be set properly for master/slave configurations to work. If your master/slave configuration does not perform as expected, use

ccAcqFifo::isValid() to check each FIFOs configuration. If **isValid()** returns false, use **ccAcqProblem::hasInvalidMaster()** or **ccAcqProblem::hasInvalidSlave()** to determine whether the FIFO's master/slave configuration is supported.

ccTriggerProp

ccUIAffineRect

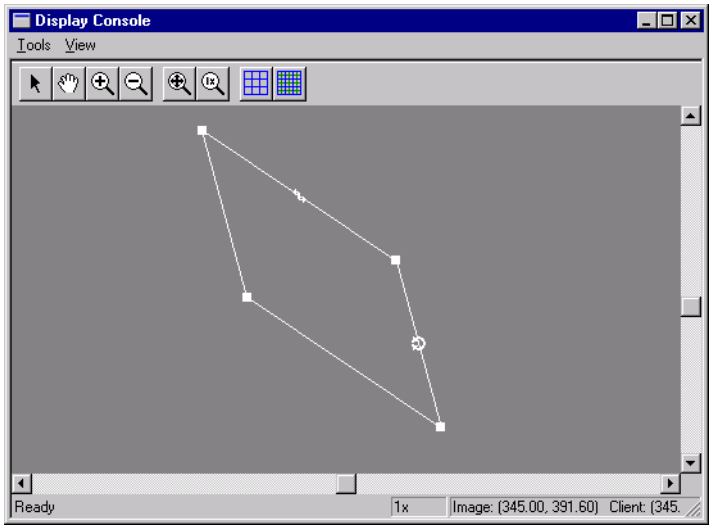
```
#include <ch_cvl/uishapes.h>

class ccUIAffineRect : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an affine rectangle (**ccAffineRectangle**) that you can manipulate. When you add a **ccUIAffineRect** object to a **ccDisplay** window an affine rectangle is displayed that can be moved about, resized, and rotated using your mouse. The following is an example of an affine rectangle that has been added to a **ccDisplayConsole** window.



ccUIAffineRect is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIAffineRect** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUIAffineRect** has four resizing handles which all operate in the same way.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the object. The rotation handle is on the right-hand edge of the rectangle.
 Skew handle	Left-click on the skew handle holding the button down, and drag the handle to change the affine rectangle skew.

After manipulating an affine rectangle in a **ccDisplay** window, you will generally want to retrieve the final affine rectangle state and use it in some way. You can use code similar to the following to retrieve the manipulated affine rectangle.

```
ccAffineRectangle new_affineRect;
new_affineRect = ui_affineRect->affRect();
```

Where **ui_affineRect** is the manipulated **ccUIAffineRect** object.

Notes

ccUIAffineRect and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIAffineRect

```
ccUIAffineRect (ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccAffineRectangle**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The rectangle's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Enumerations

eLock

This enumeration lets you specify which handles are not visible when the object is selected. By using these locks, you can prevent a shape from being modified along different degrees of freedom.

Value	Meaning
eNoLocks = 0	No locks. All manipulations allowed.
eSizeLock = 0x1	The object cannot be resized.
eRotLock = 0x2	The object cannot be rotated.
eSkewLock = 0x4	The object cannot be skewed.

Public Member Functions

affRect

```
void affRect(const ccAffineRectangle& r);

const ccAffineRectangle& affRect() const;
```

- `void affRect(const ccAffineRectangle&);`

  Associates a **ccAffineRectangle** with this manipulation object.

  Parameters

  r The affine rectangle to make manipulable.

- `const ccAffineRectangle& affRect() const;`

  Gets the **ccAffineRectangle** associated with this manipulation object.

updateHandles

```
virtual void updateHandles ();
```

Update all of my handles. Called when handles have been repositioned in the display.

showHandles `virtual void showHandles (bool show);`

Make my handles visible/invisible depending on *show*, and the DOF locks. If the handles aren't created yet, create them.

Parameters

show True = show handles. False = do not show handles.

freeHandles `virtual void freeHandles ();`

Deletes all handles.

locks

```
c_UInt32 locks() const;
void locks(c_UInt32 b);
```

- `c_UInt32 locks() const;`

  Returns the lock bits that indicate which resizing operations are not allowed for this object.

- `void locks(c_UInt32 b);`

  Sets the lock bits that indicate the resizing operations that are not allowed for this object.

  Parameters

  b The lock bits. May be any of the following ORed together:

  ccUIAffineRect::eNoLocks
  ccUIAffineRect::eSizeLock
  ccUIAffineRect::eRotLock
  ccUIAffineRect::eSkewLock

  These lock values are described in *Enumerations* on page 3163.

setLocks `void setLocks(c_UInt32 b);`

ORs the specified locks to the current locks.

Parameters

b The lock bits. May be any of the following ORed together:

ccUIAffineRect::eNoLocks
ccUIAffineRect::eSizeLock
ccUIAffineRect::eRotLock
ccUIAffineRect::eSkewLock

These lock values are described in *Enumerations* on page 3163.

clrLocks `void clrLocks(c_UInt32 b);`

Clears the locks specified in *b*. (ANDs the current locks with the complement of *b*)

Parameters

b The lock bits. May be any of the following ORed together:

ccUIAffineRect::eNoLocks
ccUIAffineRect::eSizeLock
ccUIAffineRect::eRotLock
ccUIAffineRect::eSkewLock

These lock values are described in *Enumerations* on page 3163.

isTouched `virtual bool isTouched (const cc2Vect& pt, double scale);`

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt The position to test in the coordinate system used to draw the shape.

scale The scale to convert *pt* to pixels.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;
virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the center location of the affine rectangle relative to its parent, in tablet coordinates.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the center location of the affine rectangle relative to its parent, in tablet coordinates.

  Parameters

  pos The new center location.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the affine rectangle in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

Static Functions

touchDist

```
static float touchDist();
static void touchDist(float d);
```

- `static float touchDist();`

Gets the distance from all instance of this object that is still considered a click on the object. The distance is in display coordinates.

- `static void touchDist (float d);`

Sets the distance from all instance of this object that is still considered a click on the object. The distance is in display coordinates.

Parameters

d	The distance from the object.

ccUICircle

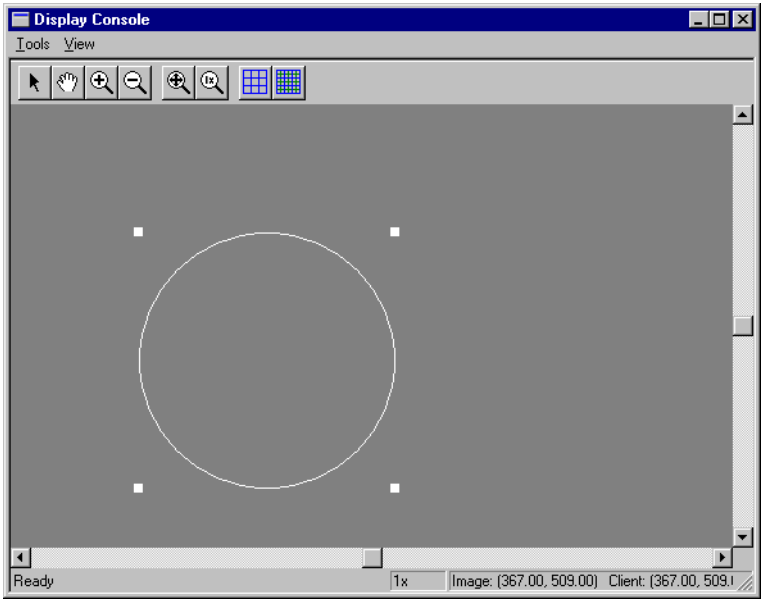
```
#include <ch_cvl/uishapes.h>

class ccUICircle : public ccUIGenRect;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a circle (**ccCircle**) that you can manipulate. When you add a **ccUICircle** object a **ccDisplay** window a circle is displayed that can be moved about and resized using your mouse. The following is an example of a circle that has been added to a **ccDisplayConsole** window:



ccUICircle is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUICircle** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUICircle** has four resizing handles which all operate in the same way.

After manipulating a circle in a **ccDisplay** window, you will generally want to retrieve the final circle state and use it in some way. You can use code similar to the following to retrieve the manipulated circle.

```
ccCircle new_circle;
new_circle = ui_circle->circle();
```

Where **ui_circle** is the manipulated **ccUICircle** object.

Notes

ccUICircle and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUICircle

```
ccUICircle(ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccCircle**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent	The circle's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

circle

```
void circle(const ccCircle& c);
ccCircle circle() const;
```

- Associates a **ccCircle** with this manipulation object.

  Parameters

  c The circle to make manipulable.

- `ccCircle circle() const;`

  Gets the **ccCircle** associated with this manipulation object.

ccUICircle

ccUICoordAxes

```
#include <ch_cvl/uishapes.h>

class cmImport_cogdisp ccUICoordAxes : public ccUIManShape;
```

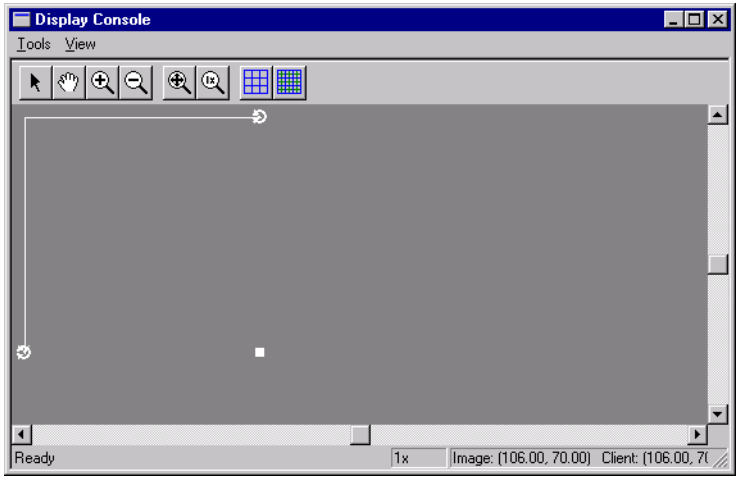
Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an x,y coordinate frame (**ccCoordAxes**) that you can manipulate. When you add a **ccUICoordAxes** object to a **ccDisplay** window an x,y coordinate frame is displayed that can be moved about, resized, and rotated using your mouse. You can lock specific degrees of freedom of the coordinate frame such as skew and rotation so that they cannot be changed with the mouse. See **ccUICoordAxes::locks()**.

The displayed coordinate frame is represented by a **cc2Xform** that maps user coordinates to client-specified base coordinates. Base coordinates are defined by a **cc2Xform** that maps base coordinates to tablet coordinates (*eDisplayCoords*, *eImageCoords*, or *eClientCoords* for **ccDisplay** objects). The full transform from user coordinates to tablet coordinates is obtained by multiplying the **ccUICoordAxes::tabletFromBase()** and **ccUICoordAxes::baseFromUser()** transforms.

The following is an example of a **ccUICoordAxes** coordinate frame that has been added to a **ccDisplayConsole** window.



ccUICoordAxes is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUICoordAxes** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the **ccUICoordAxes** object.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the object. Each axis has its own rotation handle. The axes rotate independently.

After manipulating a coordinate frame in a **ccDisplay** window, you will generally want to retrieve the final coordinate frame state and use it in some way. You can use code similar to the following to retrieve the manipulated coordinate frame.

```
ccCoordinageAxes new_coordAxes;
new_coordAxes = ui_coordAxes->coordAxes();
```

Where **ui_coordAxes** is the manipulated **ccUICoordAxes** object.

Notes

ccUICoordAxes and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUICoordAxes

```
ccUICoordAxes (ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccCoordAxes** with an identity transformation, default locks, and default axes with length 0.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The coordinate axes parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Enumerations

eLock

The following enumerations let you specify which handles are not visible when the object is selected. By using these locks, you can prevent the object from being modified along different degrees of freedom. You can OR these values together to lock multiple degrees of freedom. For example, to prevent resizing of any of the axes, call **locks(ccUICoordAxes::eScaleLock | ccUICoordAxes::eAxesLenLock)**.

Value	Meaning
eNoLocks = 0	No locks are set. All manipulations allowed.
ePosLock = 0x1	Locks position. *Dragable* is FALSE when position is locked.
eScaleLock = 0x2	Locks scale. The object cannot be scaled. If scale is locked, aspect ratio is locked. Applies only to the X and Y coordinate axes adjustment handles, preventing scaling of the axes.
eAspectLock = 0x4	Locks aspect ratio. The object keeps the same aspect ratio when resized. The aspect ratio is locked regardless of *aspectLock* flag.
eRotLock = 0x8	Locks rotation. The object cannot be rotated. If rotation is locked, skew is locked
eSkewLock = 0x10	Locks skew. Object cannot be skewed. Skew is locked regardless of *skewLock* flag
eAxesLenLock = 0x20	Locks axes length independent of scale lock. Applies only to the coordinate axes size handles, preventing resizing of the axes.
eAllLock = 0x3F	All locks are set. No manipulations allowed

Public Member Functions

baseFromUser

```
const cc2Xform& baseFromUser () const;
void baseFromUser (const cc2Xform&);
```

- `const cc2Xform& baseFromUser () const;`

  Returns the base coordinates from user coordinates transformation object.

- `void baseFromUser (const cc2Xform& form);`

  Sets the base coordinates from user coordinates transformation object.

  Parameters

  form The transformation object

tabletFromBase

```
const cc2Xform& tabletFromBase () const;
void tabletFromBase (const cc2Xform&);
```

- `const cc2Xform& tabletFromBase () const;`

  Returns tablet coordinates from base coordinates transformation object.

- `void tabletFromBase (const cc2Xform& form);`

  Sets tablet coordinates from base coordinates transformation object.

  Parameters

  form The transformation object.

tabletFromUser

```
const cc2Xform& tabletFromUser () const;
```

Returns the full tablet coordinates from user coordinates transformation object. This is equivalent to multiplying the **tabletFromBase()** and **baseFromUser()** transformation objects.

axesLen

```
double axesLen () const;

void axesLen (double len);
```

- `double axesLen () const;`

  Returns the length of the displayed coordinate axes.

- `void axesLen (double len);`

  Sets the length of the displayed coordinate axes.

  Parameters

len	*len* > 0, sets length in user coordinates. *len*= 0: sets length to default value (-50). *len* < 0: sets length of x axis in tablet coordinates to the absolute of the value given. The length of y axis is determined by the aspect ratio of transform.

  Notes

  When the user manipulates the coordinate axes length by dragging the appropriate handle, **axesLen()** will be set positive in user coordinates, regardless of previous setting.

axesULen

```
double axesULen () const;
```

Returns the displayed length of the coordinate axes in user coordinates.

coordAxes

```
void coordAxes(const ccCoordAxes& ax, double xLen = 1,
  double yLen = 1);

const ccCoordAxes& coordAxes() const;
```

- `void coordAxes(const ccCoordAxes& axes, double xLen = 1, double yLen = 1);`

  Sets the coordinate axes representation in tablet coordinates. The transforms are modified, **tabletFromBase()** is set to the identity transform, and **baseFromUser(**) is computed from the coordinate axes and the specified x and y axes lengths in user coordinates. The axes lengths are used to compute x and y scale.

  Parameters

axes	The coordinate axes object.
xLen	Length in x direction.

yLen Length in y direction.

Notes

This setter requires that the coordinate *axes* input not be degenerate.

- `const ccCoordAxes& coordAxes() const;`

Returns the coordinate axes representation in tablet coordinates.

updateHandles `virtual void updateHandles ();`

Tells all handles to update themselves because of a change.

showHandles `virtual void showHandles (bool b);`

Make handles visible or invisible. If the handles aren't created yet, this function creates them.

Parameters

b True sets handles to visible. False sets handles to invisible.

freeHandles `virtual void freeHandles ();`

Deletes all handles.

locks

```
c_UInt32 locks () const;
void locks (ccUICoordAxes::eLocks = b);
```

- `c_UInt32 locks () const;`

Returns the lock bits that indicate which resizing operations are not allowed for this object.

- `void locks (c_UInt32 b);`

Sets the lock bits that indicate the resizing operations that are not allowed for this object.

Parameters

b The lock bits. May be any of the following ORed together:

ccUICoordAxes::eNoLocks
ccUICoordAxes::ePosLock
ccUICoordAxes::eScaleLock

ccUICoordAxes::eRotLock
ccUICoordAxes::AspectLock
ccUICoordAxes::SkewLock
ccUICoordAxes::AxesLenLock
ccUICoordAxes::AllLock

These lock values are described on page 3173

setLocks `void setLocks (c_UInt32 b);`

ORs the specified locks to the current locks.

Parameters

b The locks to OR with the current locks. May be any of the following ORed together:

ccUICoordAxes::eNoLocks
ccUICoordAxes::ePosLock
ccUICoordAxes::eScaleLock
ccUICoordAxes::eRotLock
ccUICoordAxes::AspectLock
ccUICoordAxes::SkewLock
ccUICoordAxes::AxesLenLock
ccUICoordAxes::AllLock

These lock values are described on page 3173.

clrLocks `void clrLocks (c_UInt32 b);`

Clears the locks specified in *b*. (ANDs the current locks with the complement of *b*)

Parameters

b The locks to clear. May be any of the following ORed together:

ccUICoordAxes::eNoLocks
ccUICoordAxes::ePosLock
ccUICoordAxes::eScaleLock
ccUICoordAxes::eRotLock
ccUICoordAxes::AspectLock
ccUICoordAxes::SkewLock
ccUICoordAxes::AxesLenLock
ccUICoordAxes::AllLock

These lock values are described on page 3173.

isTouched

```
virtual bool isTouched (const cc2Vect& pt,
  double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;

virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the position of the coordinate frame origin relative to its parent, in tablet coordinates.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the position of the coordinate frame origin relative to its parent, in tablet coordinates.

  Parameters

pos	The new position.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the coordinate frame in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.

m The drawing mode. Must be one of:
ccUIShapes::drawNormal
ccUIShapes::drawAbridged
ccUIShapes::drawDrag

Static Functions

touchDist

```
static float touchDist();
static void touchDist(float d);
```

- `static float touchDist();`

  Gets the distance from all instances of this object that is still considered to be a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

  Sets the distance from all instances of this object that is still considered to be a click on the object. The distance is specified in display coordinates.

  Parameters

  d The distance from the object.

ccUICoordAxes

ccUIEllipse

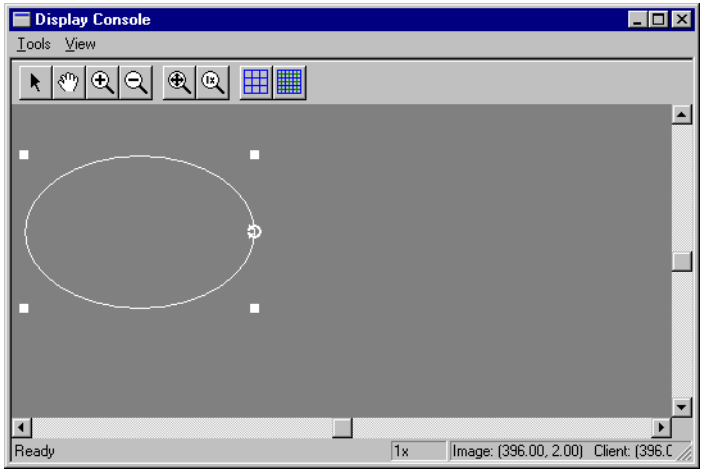
```
#include <ch_cvl/uishapes.h>

class ccUIEllipse : public ccUIGenRect;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an ellipse (**ccEllipse2**) that you can manipulate. When you add a **ccUIEllipse** to a **ccDisplay** window an ellipse is displayed that can be moved about, reshaped, resized, and rotated using your mouse. The following is an example of an ellipse that has been added to a **ccDisplayConsole** window:



ccUIEllipse is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the

ccUIObject and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIEllipse** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUIEllipse** has four resizing handles which all operate in the same way.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the ellipse. The rotation handle is on the ellipse right-hand edge.

After manipulating an ellipse in a **ccDisplay** window, you will generally want to retrieve the final ellipse state and use it in some way. You can use code similar to the following to retrieve the manipulated ellipse.

```
ccEllipse new_ellipse;
new_ellipse = ui_ellipse->ellipse();
```

Where **ui_ellipse** is the manipulated **ccUIEllipse** object.

Notes

ccUIEllipse and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIEllipse

```
ccUIEllipse(ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccEllipse2**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The ellipse's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

ellipse2

```
void ellipse2(const ccEllipse2&);
ccEllipse2 ellipse2() const;
```

- `void ellipse2(const ccEllipse2& e);`

  Associates a **ccEllipse2** with this manipulation object.

  Parameters

  e The ellipse to make manipulable.

- `ccEllipse2 ellipse2() const;`

  Returns the **ccEllipse2** associated with this manipulation object

Deprecated Members

The following functions are deprecated and are provided for backwards compatibility only. Use the functions above instead.

ellipse

```
void ellipse(const ccEllipse& e);
ccEllipse ellipse() const;
```

- `void ellipse(const ccEllipse& e);`

  Associates a **ccEllipse** with this manipulation object.

  Parameters

  e The ellipse to make manipulable.

- `ccEllipse ellipse() const;`

  Gets the **ccEllipse** associated with this manipulation object.

ccUIEllipse

ccUIEllipseAnnulusSection

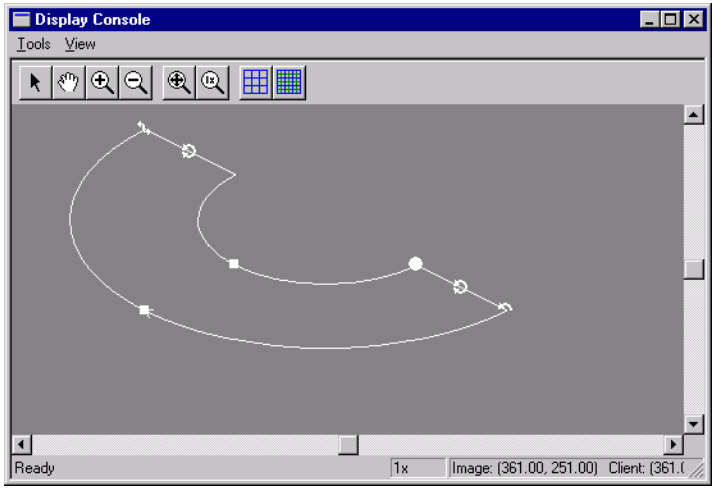
```
#include <ch_cvl/uishapes.h>

class ccUIEllipseAnnulusSection : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an elliptical annulus section (**ccEllipseAnnulusSection**) that you can manipulate. When you add a **ccUIEllipseAnnulusSection** to a **ccDisplay** window an elliptical annulus section is displayed that can be moved about, reshaped, resized, and rotated using your mouse. The following is an example of an elliptical annulus section that has been added to a **ccDisplayConsole** window:



ccUIEllipseAnnulusSection is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window

changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIEllipseAnnulusSection** uses the following handles:

Handle	Description
 Resizing handle	Left-click on a resize handle holding the button down and drag the handle to resize the ellipse radii. **ccUIEllipseAnnulusSection** has two resizing handles, one for the inner ellipse and one for the outer ellipse. When you resize an ellipse, its shape does not change.
 Subtended angle handle	Left-click on the subtended angle handle holding the button down and drag the handle to change the angle that defines the ellipse annulus section end points. **ccUIEllipseAnnulusSection** has two subtended angle handles, one for each end of the ellipse annulus section.
 Orientation handle	Left-click on the orientation handle holding the button down and drag the handle to rotate the ellipse annulus section about the common ellipse center. Note that this moves the section definition around the annulus so that when you stop at a different place, you will see a different annulus section with possibly a different shape.
 Spin handle	Left-click on a spin handle holding the button down and drag the handle to rotate the ellipse annulus section about its own center. Note that you are spinning the defined section and while it changes orientation, it does not change shape.
 Warp handle	Left-click on the warp handle holding the button down and drag the handle to warp the ellipse annulus section.

After manipulating an ellipse annulus section in a **ccDisplay** window, you will generally want to retrieve the final section state and use it in some way. You can use code similar to the following to retrieve the manipulated ellipse.

```
ccEllipseAnnulusSection new_ellipseAnnulusSection;
new_ellipseAnnulusSection =
             ui_ellipseAnnulusSection ->ellipseAnnulusSection();
```

Where **ui_ellipseAnnulusSection** is the manipulated **ccUIEllipseAnnulusSection** object.

Notes

ccUIEllipseAnnulusSection and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIEllipseAnnulusSection

```
ccUIEllipseAnnulusSection (ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccEllipseAnnulusSection**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The section's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Enumerations

eLock

This enumeration lets you specify which handles are not visible when the object is selected. By using these locks, you can prevent a section from being modified along different degrees of freedom. These values can be ORed together to form a single bit field argument to **locks()**, **setlocks()**, and **clrlocks()**.

Value	Meaning
eNoLocks = 0	No locks. All manipulations are allowed.
eRadiiLock = 0x1	The section's radii cannot be resized.
eSubtendedAngleLock = 0x2	The angle from the center of the ellipses, subtended by the section's starting and ending angles, cannot be resized.
eOrientationLock = 0x4	Disables the handle that allows rotation of the annulus section around the center of its defining ellipses.

Value	Meaning
eSpinLock = 0x8	Disables the handle that allows rotation of the annulus section around its own center.
eWarpLock = 0x10	Disables the handle that allows warping of the annulus section. Warping keeps the section approximately the same area and with approximately the same center, but changes its curvature.

Public Member Functions

ellipseAnnulusSection

```
void ellipseAnnulusSection (const ccEllipseAnnulusSection
  &e);

const ccEllipseAnnulusSection ellipseAnnulusSection()
  const;
```

- ```
  void ellipseAnnulusSection (const ccEllipseAnnulusSection
    &e);
  ```

  Associates a **ccEllipseAnnulusSection** with this manipulation object.

  Parameters

  e The ellipse annulus section to make manipulable.

  Notes

  The two ellipses used to define an ellipse annulus section must be concentric. You can define an ellipse annulus section where the two ellipses are not concentric and the code will compile with no errors. However, at run time you will get a *ccShapesError::BadGeom* error.

- ```
  const ccEllipseAnnulusSection ellipseAnnulusSection()
    const;
  ```

  Gets the **ccEllipseAnnulusSection** associated with this manipulation object.

updateHandles

```
virtual void updateHandles();
```

Calls base class method and **update()** on the rotate handle.

showHandles

```
virtual void showHandles (bool b);
```

Creates handles if they weren't already created. Calls base class method and makes handles visible or invisible depending on the flag passed in.

Parameters

b True makes handles visible. False makes handles invisible.

freeHandles

```
virtual void freeHandles();
```

Deletes the handles so that the object can not be manipulated.

locks

```
c_UInt32 locks() const;
void locks(c_UInt32 b);
```

- `c_UInt32 locks() const;`

  Returns the lock bits that indicate which resizing operations are not allowed for this object.

- `void locks(c_UInt32 b);`

  Sets the lock bits that indicate the resizing operations that are disallowed for this object.

  Parameters

  b The lock bits. May be any of the members of the **ccEllipseAnnulusSection::eLock** enumeration, ORed together

setLocks

```
void setLocks (c_UInt32 b);
```

ORs the specified locks to the current locks.

Parameters

b The lock bits to OR with the current lock bits. May be any of the members of the **ccEllipseAnnulusSection::eLock** enumeration, ORed together

clrLocks

```
void clrLocks (c_UInt32 b);
```

Clears the locks specified in *b*. (ANDs the current locks with the complement of *b*)

Set/clear lock bits.

Parameters

b The lock bits to clear. May be any of the members of the **ccEllipseAnnulusSection::eLock** enumeration, ORed together

maxRadius

```
double maxRadius() const;
void maxRadius (double maxRadius);
```

- `double maxRadius() const;`

  Returns the current maximum radius setting.

- `void maxRadius (double maxRadius);`

  Sets a value for the largest radius to which an ellipse annulus section can be resized.

  maxRadius The maximum radius size to allow.

isTouched

```
virtual bool isTouched (const cc2Vect& pt, double scale);
```

Parameters

pt The position to test in the coordinate system used to draw the shape.

scale The scale to convert *pt* to pixels.

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call this function yourself.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;
virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the position of the UI shape relative to its parent, in tablet coordinates. The position is the center of the ellipse annulus section.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the position of the UI shape relative to its parent, in tablet coordinates.

  Parameters

pos	The new position.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

Static Functions

touchDist

```
static float touchDist();

static void touchDist (float d);
```

- `static float touchDist();`

  Gets the distance from all instances of this object that is still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

  Sets the distance from all instances of this object that is still considered a click on the object. The distance is specified in display coordinates.

  Parameters

d	The distance from the objects.

ccUIEllipseAnnulusSection

ccUIEventProcessor

```
#include <ch_cvl/uievent.h>

class ccUIEventProcessor;
```

Class Properties

Copyable	No
Derivable	Not intended
Archiveable	Complex

This class provides the interface between keyboard and mouse events, and **ccUIObject** objects. There is one instance of this class for each **ccUIRootObject**.

Constructors/Destructors

ccUIEventProcessor

```
ccUIEventProcessor (ccUIRootObject* root);
```

Parameters

root Root must not be NULL.

Public Member Functions

processEvent

```
void processEvent(const ccMouseEvent& event);

void processEvent(const ccKeyboardEvent& event);
```

- ```
  void processEvent(const ccMouseEvent& event);
  ```

  Process mouse events by calling the appropriate **ccUIObject** virtual methods to implement the GUI behavior.

  Parameters

  event A mouse event.

- ```
  void processEvent(const ccKeyboardEvent& event);
  ```

  Process keyboard events by calling appropriate **ccUIObject** virtual method to implement the GUI behavior.

Parameters

event A keyboard event.

root

```
ccUIRootObject* root();
```

Returns the root UI object associated with this event processor.

owner

```
ccUIObject* owner ();
```

Returns the mouse owner. For example, which **ccUIObject** was the mouse pointing to when the mouse button was clicked.

owner() returns NULL if the **ccUIObject** mouse owner object is deleted.

anchor

```
const ccIPair& anchor () const;
```

Returns the point in screen coordinates where the mouse button down-click (not double-click) occurred. For example, a **click_()** override could call this function to determine where the click occurred, if your application needs to know.

button

```
c_UInt32 button ();
```

Returns which button owns the mouse. See **ccMouseEvent::Button** for enums.

dblClk

```
bool dblClk ();
```

Returns true if the last event was a double-click. This method is generally used in processing an up-click to distinguish those following down-clicks from those following double-clicks.

dwellCount

```
c_Int32 dwellCount ();
```

Returns the number of dwell events that have occurred since the down-click that started it.

invalidateObject

```
void invalidateObject (ccUIObject* obj);
```

Called when an object is deleted or when its root is changed so we can clean up any pointers to it or its ancestors. This is how we make sure, for example, that death does not result when someone double-clicks an object whose response to single-click is to delete itself.

Parameters

obj The object to invalidate.

mouseCapture

```
ccUIObject* mouseCapture ();
void mouseCapture (ccUIObject* obj);
```

The mouse can be associated with (captured) only one object at a time. If the root captures the mouse, the behavior will be the same as if no object has captured the mouse, and **mouseCapture()** will return the root instead of NULL.

- `ccUIObject* mouseCapture ();`

  Returns the object that has captured the mouse, or NULL if there is none.

- `void mouseCapture (ccUIObject* obj);`

  The specified object captures the mouse. NULL means release a captured mouse.

  Parameters

  obj The specified object.

lastMousePos

```
const ccIPair& lastMousePos() const;
```

Returns the last mouse position. Initially set to (0,0).

dragUpdating

```
void dragUpdating(bool b);
bool dragUpdating() const;
```

When set true, **ccUIRootObject::globalUpdate()** is called while dragging objects to update the display. When set false, the call is not made.

- `void dragUpdating(bool b);`

  Sets the update flag, true or false.

  Parameters

  b The new update flag.

- `bool dragUpdating() const;`

  Returns the current update flag.

multiSelectKey

```
void multiSelectKey(ccKeyboardEvent::VirtualKey k);

ccKeyboardEvent::VirtualKey multiSelectKey() const;
```

Specifies the keyboard key to use for multiple selections. If the key specified is held down you can make multiple selections with the mouse.

Valid values are *ccKeyboardEvent::eShift*, *ccKeyboardEvent::eControl*, and *ccKeyboardEvent::eNoVKey*. When you specify *ccKeyboardEvent::eNoVKey* multiple selections are disabled.

The default is *ccKeyboardEvent::eShift*.

- `void multiSelectKey(ccKeyboardEvent::VirtualKey k);`

Set a new key.

Parameters

k The new key.

Throws

ccUIError::BadKeyValue
If *k* is invalid.

Notes

Since each key can only have one function assigned to it at a time, the setting for **multiSelectKey()** interacts with the setting for **panKey()**. If **multiSelectKey()** is changed to the same value as **panKey()**, **panKey()** will be set to *ccKeyboardEvent::eNoVKey*.

- `ccKeyboardEvent::VirtualKey multiSelectKey() const;`

Returns the current key.

panKey

```
void panKey(ccKeyboardEvent::VirtualKey k);

ccKeyboardEvent::VirtualKey panKey() const;
```

Specifies the keyboard key to use for panning. If the specified key is held down the mouse pointer changes to a hand. You can then press the left mouse button and drag the image center to a new location.

Valid values are *ccKeyboardEvent::eShift*, *ccKeyboardEvent::eControl*, and *ccKeyboardEvent::eNoVKey*. When you specify *ccKeyboardEvent::eNoVKey* panning is disabled.

The default is *ccKeyboardEvent::eControl*.

- `void panKey(ccKeyboardEvent::VirtualKey k);`

  Set a new key.

  Parameters

  k The new key.

  Throws

  ccUIError::BadKeyValue
  If *k* is invalid.

  Notes

  Since each key can only have one function assigned to it at a time, the setting for **panKey()** interacts with the setting for **multiSelectKey()**. If **panKey()** is changed to the same value as **multiSelectKey()**, **multiSelectKey()** will be set to *ccKeyboardEvent::eNoVKey*.

- `ccKeyboardEvent::VirtualKey panKey() const;`

  Returns the current key.

ccUIEventProcessor

ccUIFormat

```
#include <ch_cvl/uistring.h>

class ccUIFormat;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Simple

This class is describes the font and alignment of string in classes that draw text such as labels.

Constructors/Destructors

ccUIFormat

```
ccUIFormat(c_Int32 fontId = -1,
  Alignment alignment = eTopLeft);
```

Creates a format object with the specified font and alignment.

Parameters

fontId — The font id to use for this format. The value –1 indicates the system font. Other values correspond to entries in the font table set in the platform-dependent display class such as **ccWin32Display::fontTable()**.

alignment — The alignment of the text. Must be one of:

ccUIFormat::eTopLeft
ccUIFormat::eTopCenter
ccUIFormat::eTopRight
ccUIFormat::eCenterLeft
ccUIFormat::eCenter
ccUIFormat::eCenterRight
ccUIFormat::eBottomLeft
ccUIFormat::eBottomCenter
ccUIFormat::eBottomRight

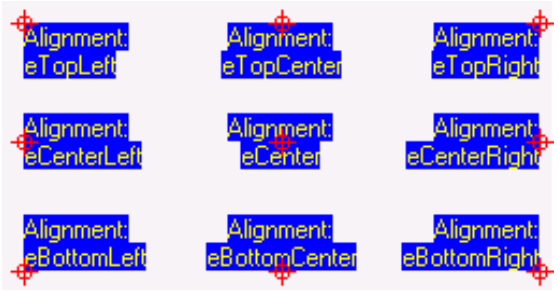
Enumerations

Alignment

```
enum Alignment;
```

These values let you specify how the text is aligned with respect to the drawing point.

Value	Meaning
eTopLeft	The following figure shows how each of the alignment values aligns the text with respect to the drawing point (indicated here by a point icon).
eTopCenter	
eTopRight	
eCenterLeft	
eCenter	
eCenterRight	
eBottomLeft	
eBottomCenter	
eBottomRight	



Public Member Functions

fontId

```
void fontId(c_Int32 id);
c_Int32 fontId() const;
```

- `void fontId(c_Int32 id);`

  Sets the font associated with this format.

  Parameters

  id — The font id to use for this format. The value –1 indicates the system font. Other values correspond to entries in the font table set in the platform-dependent display class such as **ccWin32Display::fontTable()**.

- `c_Int32 fontId() const;`

  Get the font id for this format.

alignment

```
void alignment(Alignment a);
Alignment alignment() const;
```

- `void alignment(Alignment a);`

  Specifies how the text should be aligned with respect to its drawing point.

  Parameters

  a The alignment of the text. Must be one of:

  ccUIFormat::eTopLeft
  ccUIFormat::eTopCenter
  ccUIFormat::eTopRight
  ccUIFormat::eCenterLeft
  ccUIFormat::eCenter
  ccUIFormat::eCenterRight
  ccUIFormat::eBottomLeft
  ccUIFormat::eBottomCenter
  ccUIFormat::eBottomRight

- `Alignment alignment() const;`

  Gets the alignment associated with this format.

ccUIFormat

ccUIGDShape

```
#include <uigdbase.h>

class ccUIGDShape : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This is an abstract base class that provides common support for all manipulable geometrical description shapes. These shapes have the following properties:

- They implement geometric descriptions of objects in shape coordinate space. The shape coordinate frame is attached to the shape and does not change when the shape is dragged.
- They embody a linear transformation that maps shape coordinates to model coordinates. Such linear transformation is automatically updated as the shape is dragged.
- The drawing is with respect to model coordinate space.
- They provide base class support for copying.
- The changes to a shape can be undone.
- Dragging can cause rotation, scaling or translation depending on the selected dragging mode.

Constructors/Destructors

ccUIGDShape

```
ccUIGDShape (ccUIObject* parent = NULL);
```

Constructs this **ccUIGDShape** with parent frame *parent*.

Parameters

parent — The parent frame of this **ccUIGDShape**. Typically, the display system will set up the parent for you when you add the shape to the display. In this case, the default (NULL) should be used.

Enumerations

ceDragMode `enum ceDragMode`

This enumeration defines the dragging modes of this **ccUIGDShape**.

Value	Meaning
eTranslate	Dragging translates this **ccUIGDShape**
eRotate	Dragging rotates this **ccUIGDShape**
eScale	Dragging scales this **ccUIGDShape**

Public Member Functions

clone

```
virtual ccUIGDShape* clone() const = 0;
```

Returns a pointer to a copy of this **ccUIGDShape** shape.

modelFromShape

```
virtual cc2Rigid modelFromShape() const = 0;
virtual void modelFromShape(const cc2Rigid& mFp) = 0;
```

- `virtual cc2Rigid modelFromShape() const = 0;`

  Returns the transformation that maps shape coordinates to model coordinates (typically, the model coordinates are the client coordinates of your application).

- `virtual void modelFromShape(const cc2Rigid& mFp) = 0;`

  Sets the transformation that maps shape coordinates to model coordinate to *mFp*.

  Parameters

  mFp — The transformation that maps shape coordinates to model coordinates.

undo

```
virtual void undo() = 0;
```

Undoes the last action that changed this **ccUIGDShape**.

condChanged

```
virtual bool condChanged() = 0;
```

Returns true if this **ccUIGDShape** has changed since the last time this function was called. Returns false otherwise.

Notes

The function will return false if this **ccUIGDShape** is in the default constructed state.

changed

```
virtual bool changed() const = 0;
```

Returns true if this **ccUIGDShape** shape has changed since the last time **ccUIGenPoly::condChanged** was called. Returns false otherwise.

Notes

The function will return false if this **ccUIGDShape** is in the default constructed state.

dragMode

```
void dragMode(ceDragMode mode);
ceDragMode dragMode() const;
```

- `void dragMode(ceDragMode mode);`

  Sets the dragging mode of this **ccUIGDShape** to *mode*.

  Parameters

  mode The dragging mode this **ccUIGDShape** is set to.

- `ceDragMode dragMode() const;`

  Gets the current dragging mode of this **ccUIGDShape**.

dragOrigin

```
void dragOrigin(const cc2Vect& origin);
const cc2Vect& dragOrigin() const;
```

- `void dragOrigin(const cc2Vect& origin);`

  Sets the origin in model coordinates about which this **ccUIGDShape** is rotated or scaled when **useDragOrigin()** is true.

  Parameters

  origin The origin about which this **ccUIGDShape** is rotated or scaled.

  Notes

  The origin remains unchanged after dragging.

- `const cc2Vect& dragOrigin() const;`

  Gets the current origin in model coordinate space about which this shape will rotate or scale when **useDragOrigin()** is true.

  Notes

  When **useDragOrigin()** is false, dragging is about the bounding box of this **ccUIGDShape** computed in model coordinates.

useDragOrigin

```
bool useDragOrigin() const;

void useDragOrigin(bool use);
```

- `bool useDragOrigin() const;`

  Returns true if this **ccUIGDShape** is in drag origin mode.

- `void useDragOrigin(bool use);`

  When *use* is true, **ccUIGDShape** is in drag origin mode. The rotation and scale dragging is about the origin set by **ccUIGDShape::dragOrigin()**.

  Parameters

use	The boolean variable that controls whether this **ccUIGDShape** is in drag origin mode.

drawWithXform

```
virtual void drawWithXform (ccUITablet& t,
  const ccColor& c,
  const cc2Xform& tabletFromModel,
  ccUIShapes::DrawMode m = drawNormal) = 0;
```

Draws this **ccUIGDShape** using the tablet *t* in the color specified by *c*. Before drawing, the shape is mapped to tablet coordinates by using *tabletFromModel*.

Parameters

t	The tablet used to draw this **ccUIGDShape**.
c	The color used to draw this **ccUIGDShape**.
tabletFromModel	The transformation that maps model coordinates to tablet coordinates.
m	The drawing mode.

snapAngle

```
void snapAngle(const ccDegree& angle);

const ccDegree& snapAngle() const;
```

- `void snapAngle(const ccDegree& angle);`

  Sets the angular step of the rotational dragging of this **ccUIGDShape** to *angle*. After setting this value, this **ccUIGDShape** can be rotated only in steps of *angle* degrees. During dragging this **ccUIGDShape** is snapped to the closest angle compatibile with this constraint.

  Parameters

  angle The smallest angle the shape can be rotated.

- `const ccDegree& snapAngle() const;`

  Returns the angular step used for the rotational dragging of this **ccUIGDShape**.

boundingBox

```
virtual ccRect boundingBox(
  const cc2Xform& clientFromShape) const = 0;

virtual ccRect boundingBox(
  const cc2Rigid& clientFromShape) const;

virtual ccRect boundingBox() const;
```

- ```
  virtual ccRect boundingBox(
    const cc2Xform& clientFromShape) const = 0;
  ```

  Returns the bounding rectangle of this **ccUIGDShape** in the client coordinates specified by *clientfromShape*.

  Parameters

  clientFromShape The transformation that maps shape coordinates to client coordinates.

- ```
  virtual ccRect boundingBox(
    const cc2Rigid& clientFromShape) const;
  ```

  Returns the bounding rectangle of this **ccUIGDShape** in the client coordinates specified by the rigid transformation *clientFromShape*.

  Parameters

  clientFromShape The rigid transformation that maps shape coordinates to client coordinates.

Notes

The client coordinates are defined by a rigid transformation. This type of transformation does not allow you to model scale changes between shape and client coordinates (see *Math Foundations of Transformations* in the *CVL User's Guide*).

- `virtual ccRect boundingBox() const;`

Returns the bounding rectangle of this **ccUIGDShape** in model coordinates.

snappedRigid

```
static cc2Rigid snappedRigid (const cc2Rigid& r,
  const ccDegree& angle);
```

Returns the closest rigid transformation to *r* that is compatibile with the angular step *angle*.

Parameters

r	The rigid transformation.
angle	The angular step.

snappedAngle

```
static ccDegree snappedAngle (const ccDegree& a,
  const ccDegree& angle);
```

Returns the closest angle to *a* that is compatibile with the angular step *angle*.

Parameters

a	The angle.
angle	The angular step.

axesColor

```
void axesColor(const ccColor &c);

const ccColor& axesColor() const;
```

Determines the displayed coordinate axes color.

The default color is yellow.

- `void axesColor(const ccColor &c);`

Sets the axes color. The axes color changes immediately.

Parameters

c	The new color.

- `const ccColor& axesColor() const;`

  Returns the current axes color.

axesVisible

```
void axesVisible(bool bVal);
bool axesVisible();
```

Determines if the coordinate axes are visible on the display. When set to true, the axes are visible. When set to false, the axes are not visible.

The default setting is true.

- `void axesVisible(bool bVal);`

  Set the visible flag, true or false. The axes are erased or drawn immediately.

  Parameters

  bVal The new flag value.

- `bool axesVisible();`

  Returns the current visible flag.

ccUIGenAnnulus

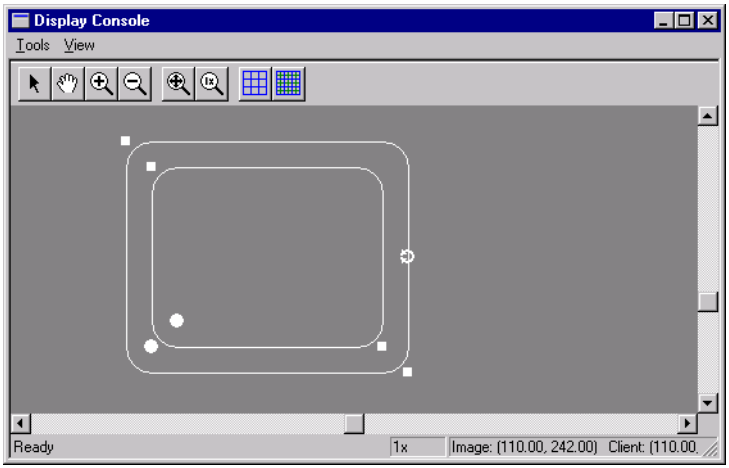
```
#include <ch_cvl/uishapes.h>

class ccUIGenAnnulus : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a generalized annulus (**ccGenAnnulus**) that you can manipulate. When you add a **ccUIGenAnnulus** object to a **ccDisplay** window a generalized annulus is displayed that can be moved about, resized, and rotated using your mouse. The following is an example of a generalized annulus that has been added to a **ccDisplayConsole** window.



ccUIGenAnnulus is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the

ccUIObject and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIGenAnnulus** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUIGenAnnulus** has two resizing handles each, for both the inner rectangle and the outer rectangle.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the annulus. The rotation handle is on the annulus right-hand edge.
 Rounded corner handle	Left-click on the rounded corner handle holding the button down, and drag the handle to the object interior to round the corners. Dragging the handle to the object exterior squares the corners. Corner rounding handles are at the lower left-hand corner of each rectangle. One handle controls corner rounding for all four rectangle corners. Corner rounding for each rectangle is performed independently.

After manipulating a generalized annulus in a **ccDisplay** window, you will generally want to retrieve the final annulus state and use it in some way. You can use code similar to the following to retrieve the manipulated annulus.

```
ccGenAnnulus new_Annulus;
new_Annulus = ui_genAnnulus->annulus();
```

Where **ui_genAnnulus** is the manipulated **ccUIGenAnnulus** object.

Notes

ccUIGenAnnulus and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIGenAnnulus

```
ccUIGenAnnulus (ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccGenAnnulus**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The annulus's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Enumerations

eSide

The following enumerations let you specify the boundary of the annulus object.

Value	Meaning
eInner	Inner boundary of annulus
eOuter	Outer boundary of annulus

Public Member Functions

annulus

```
void annulus(const ccGenAnnulus& a);
const ccGenAnnulus& annulus() const;
```

- `void annulus (const ccGenAnnulus& a);`

  Associates a **ccGenAnnulus** with this manipulation object.

  Parameters

  a The generalized annulus to make manipulable.

- `const ccGenAnnulus& annulus() const;`

  Gets the **ccGenAnnulus** associated with this manipulation object.

isTouched

```
virtual bool isTouched (const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt The position to test in the coordinate system used to draw the shape.

scale The scale to convert *pt* to pixels.

updateHandles

```
virtual void updateHandles ();
```

Calls base class method and **update()** on the rotate handle.

showHandles

```
virtual void showHandles (bool b);
```

Creates handles if they weren't already created. Calls base class method and makes handles visible or invisible depending on the flag passed in.

Parameters

b	True makes handles visible. False makes handles invisible

freeHandles

```
virtual void freeHandles ();
```

Deletes the handles so that the object can not be manipulated.

setHandle

```
void setHandle (ccUIGenAnnulusRotHandle* rh);
```

If the passed handle is non NULL, this function replaces the rotate handle.

Parameters

rh	The rotation handle object

Notes

This function is very useful for implementing constraint behavior. To implement your own constraint behavior, make a handle which derives from the appropriate handle type and implement your own constraint checking in the handle's **dragAnimate_** and **dragStop_** routines.

updateFromUIGR

```
void updateFromUIGR();
```

Updates the **ccGenAnnulus** object based on the **ccGenRect** objects of the contained **ccUIGenRect** objects. This function is called after the **ccGenRect** objects have stopped moving, which is called from the **dragStop()** routines of the handles and of this shape.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;

virtual void pos_ (const cc2Vect& pos);
```

- `virtual ccPoint pos_ () const;`

  Returns the position of the **ccUIGenAnnulus** relative to its parent, in tablet coordinates.

  This function always returns (0,0). This is necessary because the two **ccUIGenRect** objects that makeup the annulus must be children of the **ccUIGenAnnulus**, and since **absPos()** traverses the ancestor tree to find screen coordinates of the current object, the **ccUIGenRect** coordinates are offset by the position of the **ccUIGenAnnulus**. Since we want the position to be the same as if they were just regular **ccUIGenRect** objects, we need to make the returned position of the **ccUIGenAnnulus** be (0, 0).

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the position of the **ccUIGenAnnulus** relative to its parent, in tablet coordinates.

  Parameters

pos	The new position.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal;* draws a full **ccUIGenAnnulus** *ccUIShapes::drawAbridged;* draws a full **ccUIGenAnnulus** *ccUIShapes::drawDrag;* draws a **ccUIGenAnnulus** without rounding

dragStop_

```
virtual void dragStop_ (
  const ccIPair& start,
  const ccIPair& stop);
```

An override.

Calls **dragStop_()** of the inner **ccUIGenRect**. This is needed when the annulus is being dragged, instead of the inner or outer **ccUIGenRect**s.

Parameters

start	Starting drag position in screen coordinates.
stop	Ending drag position of the drag in screen coordinates.

Static Functions

touchDist

```
static float touchDist ();
static void touchDist (float d);
```

- `static float touchDist();`

Gets the distance from all instances of this object that is still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

Sets the distance from all instances of this object that is still considered a click on the object. The distance is specified in display coordinates.

Parameters

d	The distance from the objects.

checkValid

```
static bool checkValid(const ccGenRect& inner,
          const ccGenRect& outer, eSide = type);
```

Returns true if *inner* is contained in *outer* such that a flood fill starting one pixel in towards the center from any point on the outer boundary will completely color the annulus.

Parameters

inner	The rectangle comprising inner boundary of annulus.
outer	The rectangle comprising outer boundary of annulus.
type	The type of boundary. Must be one of the following:

ccUIGenAnnulus::eInner
ccUIGenAnnulus::eOuter

These values are described in Enumerations on page 3213.

ccUIGenAnnulus

ccUIGenPoly

```
#include <uignpoly.h>

class ccUIGenPoly : public virtual ccUIGDShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class is used to make a **cc2Wireframe** object manipulable from within a **ccDisplay** object. You can scale, rotate, and translate a **ccUIGenPoly** shape by using the functions provided by the base **ccUIGDShape** class. **cUIGenPoly** includes editing modes for changing shape attributes that are specific to wireframes and provides mutually exclusive drawing modes for drawing rectangles, triangles, circles and polygons.

Constructors/Destructors

ccUIGenPoly

```
ccUIGenPoly (ccUIObject* parent = NULL);
```

Constructs this **ccUIGenPoly** shape in no-edit and no-draw mode.

Parameters

parent — The parent frame of this **ccUIGenPoly** shape. Typically, the display system will set up the parent for you when you add the shape to the display. In this case, the default (NULL) should be used.

Enumerations

ceEditMode `enum ceEditMode`

This enumeration defines the editing modes of this **ccUIGenPoly** shape.

Value	Meaning
eNoEditMode	No-edit mode (default value). Only normal dragging modes are supported.
eVertexMode	Editing vertex mode. It displays and enable handles at the vertices of a polygon. You can drag the handles to change the position of the vertices.
eNewVerticesMode	Create new vertices mode. It displays and enables handles at the midpoint of each segment and at the first and last vertex if the polygon is open. You can drag these handles to insert and change the position of vertices.
eRoundingMode	Rounding vertices mode. It displays and enables handles at the midpoint of each segment and at all but the end vertices of the closed polygon. You can drag a segment handle to change its angle span. You can drag a vertex handle to change its rounding.
eVertDelMode	Vertex-deletion mode. Normal dragging is disabled and no handles are visible. You can click on vertices to delete them. If the deleted vertex is not an end vertex, the vertices on either side are directly connected by a new segment.
eSegDelMode	Segment-deletion mode. Normal dragging is disabled and no handles are visible. You can click on segments to delete them.

Notes

The **ccUIGenPoly::isRectilinear(bool r)** member function allows you to constrain this **ccUIGenPoly** shape to be rectilinear. In this case *eNewVerticesMode*, *eRoundingMode* and *eVertDelMode* are not available.

ceDrawMode `enum ceDrawMode`

This enumeration defines the drawing modes of this **ccUIGenPoly** shape.

Value	Meaning
eNoDrawMode	No-draw mode (default value). Normal editing and dragging mode
eDrawPolygonMode	Draw polygon mode. The display enters a mode in which the first mouse click action will place the first vertex, the second mouse click the second vertex, and so on, thereby creating a polygon. A right mouse click will place the last vertex of the polygon, while a double-click or a click on the first vertex will close the polygon. In either case, the object and display are returned to no-draw mode.
eDrawCircleMode	Draw circle mode. The display enters a mode in which the first mouse down action places the center of the circle, and a subsequent dragging determines the radius of the circle by the current distance of the mouse from the center. The circle is finished and the object and display are returned to no-draw mode after a mouse up action.
eDrawRectMode	Draw rectangle mode. The display enters a mode in which the first mouse down action places a corner of the rectangle, and a subsequent dragging determines the diagonal of the rectangle (the line segment between the mouse and the first corner). The rectangle is finished and the object and display are returned to no-draw mode after a mouse up action.

Value	Meaning
eDrawCornerMode	Draw corner mode. The display enters a mode in which the first mouse down action places the vertex of the 90° corner, and a subsequent dragging determines one side of the corner by the line segment between the mouse and the vertex (the other side has the same length). The corner is finished and the object and display are returned to no-draw mode after a mouse up action.
eDrawTriangleMode	Draw triangle mode. The display enters a mode in which the first mouse down action places the center of the triangle, and a subsequent dragging determines one of the base vertices of an isosceles triangle, the vertex with height of the odd vertex being determined by the distance from the base to the center. The triangle is finished and the object and display are returned to no-draw mode after a mouse up action.

Notes

The **ccUIGenPoly::isRectilinear(bool r)** member function allows you to constrain this **ccUIGenPoly** shape to be rectilinear. In this case *eDrawCircleMode* and *eDrawTriangleMode* are not available, and the vertices of polygons drawn in e*DrawPolygonMode* have angles equal to 90°.

Public Member Functions

wireframe

```
cc2Wireframe wireframe() const;
```

Returns the wireframe corresponding to this **ccUIGenPoly** shape in model coordinates (typically, the client coordinates of your application).

shape

```
void shape (const cc2Wireframe& wrfrm);
const cc2Wireframe& shape() const;
```

- `void shape (const cc2Wireframe& wrfrm);`

  Sets this **ccUIGenPoly** shape to *wrfrm*. Calling this function causes handles to be drawn.

  Parameters

  wrfrm The wireframe object this **ccUIGenPoly** shape is set to.

  Throws

  ccShapesError::BadGeom
  wrfrm is immutable.

- `const cc2Wireframe& shape() const;`

  Returns the **cc2Wireframe** object from this **ccUIGenPoly** shape.

shapeAndModelPos

```
void shapeAndModelPos (const cc2Wireframe& wrfrm,
  const cc2Rigid& mFp);
```

Sets this **ccUIGenPoly** shape to *wrfrm* and the transformation that maps shape coordinates to model coordinates to *mFp*.

Parameters

wrfrm The wireframe this **ccUIGenPoly** is set to.

mFp The **cc2Rigid** transformation that maps shape coordinates to model coordinates.

Throws

ccShapesError::BadGeom
wrfrm is immutable.

Notes

Calling **ccUIGenPoly::Undo()** after this function reverts both the wireframe and the transform.

editMode

```
ceEditMode editMode();

void editMode(ceEditMode editMode);
```

- `ceEditMode editMode();`

  Returns the editing mode of this **ccUIGenPoly** shape.

- `void editMode(ceEditMode editMode);`

  Sets the editing mode of this **ccUIGenPoly** to *editMode*.

  Parameters

  editMode The editing mode this **ccUIGenPoly** shape is set to.

  Throws

  ccShapesError::BadGeom
  This **ccUIGenPoly** shape is rectilinear and one of the following modes is selected*: ceEditMode::eNewVerticesMode, ceEditMode::eRoundingMode*, or *ceEditMode::eVertDelMode.*

  Notes

  The shape will remain the in the supplied edit mode until the mode is changed by a call to **ccUIGenPoly::editMode()**. The function has no effect if **ccUIGenPoly::isDrawing()** is true. *ceEditMode::eVertDelMode* and *ccEditMode::eSegDelMode* disable normal dragging for this **ccUIGenPoly** shape.

drawMode

```
ceDrawMode drawMode();

void drawMode(ceDrawMode drawMode,
  const ccDimTol& tol = ccDimTol());
```

- `ceDrawMode drawMode();`

  Returns the drawing mode of this **ccUIGenPoly** shape.

- `void drawMode(ceDrawMode drawMode, const ccDimTol& tol = ccDimTol());`

  Sets the drawing mode of this **ccUIGenPoly** shape to *drawMode* and sets the tolerances of the shape to *tol*. This **ccUIGenPoly** is deleted in preparation for drawing.

  Parameters

  drawMode The drawing mode of this **ccUIGenPoly** shape.

tol The tolerances to be used for each segment of this **ccUIGenPoly** shape (nominal value is ignored).

Throws

ccShapesError::BadGeom
This **ccUIGenPoly** shape is rectilinear and *ceDrawMode::eDrawCircleMode* or *ceDrawMode::eDrawTriangleMode* is selected.

Notes

The drawing mode will return to *ceDrawMode:: eNoDrawMode* after drawing has finished. Normal dragging is disabled for all but *ceDrawMode:: eNoDrawMode*.

isDrawing

```
bool isDrawing();
```

Returns true if the drawing mode of this **ccUIGenPoly** shape is different from *ceDrawMode:: eNoDrawMode*.

isRectilinear

```
void isRectilinear(bool r);
bool isRectilinear() const;
```

- `void isRectilinear(bool r);`

If *r* is true, it constrains this **ccUIGenPoly** shape to be rectilinear and snaps it to the closest rectilinear shape that has the same number of vertices. If *r* is false, it removes any rectilinear constraints. The vertices of a rectilinear **ccUIGenPoly** shape have 90° angles.

Parameters

r If *true* it forces this **ccUIGenPoly** shape to be rectilinear. If *false* it removes any rectilinear constrain.

- `bool isRectilinear() const;`

Returns true if this **ccUIGenPoly** shape is constrained to be rectilinear, false otherwise.

undo

```
virtual void undo();
```

Undoes the last action that changed this **ccUIGenPoly** shape.

modelFromShape

```
virtual cc2Rigid modelFromShape();
virtual void modelFromShape(const cc2Rigid& mFp);
```

- `virtual cc2Rigid modelFromShape();`

  Returns the transformation that maps shape coordinates to model coordinates (typically, the model coordinates are the client coordinates of your application).

- `virtual void modelFromShape(const cc2Rigid& mFp);`

  Sets the transformation that maps shape coordinates to model coordinate to *mFp*.

  Parameters

mFp	The transformation that maps shape coordinates to model coordinates.

condChanged `virtual bool condChanged();`

Returns true if this **ccUIGenPoly** shape has changed since the last time this function was called. Returns false otherwise.

Notes

The function will return false if this **ccUIGenPoly** shape is in the default constructed state.

changed `virtual bool changed() const;`

Returns true if this **ccUIGenPoly** shape has changed since the last time **ccUIGenPoly::condChanged** was called. Returns false otherwise.

Notes

The function will return false if this **ccUIGenPoly** shape is in the default constructed state.

boundingBox

```
virtual ccRect boundingBox() const;

virtual ccRect boundingBox(
  const cc2Xform& clientFromShape) const;

virtual ccRect boundingBox(
  const cc2Rigid& clientFromShape) const;
```

- `virtual ccRect boundingBox() const;`

  Returns the bounding rectangle of this **ccUIGenPoly** shape in model coordinates.

- `virtual ccRect boundingBox(`
  `const cc2Xform& clientFromShape) const;`

  Returns the bounding rectangle of this **ccUIGenPoly** shape in the client coordinates specified by *clientfromShape*.

  Parameters

  clientfromShape The transformation that maps shape coordinates to client coordinates

- `virtual ccRect boundingBox(`
  `const cc2Rigid& clientFromShape) const;`

  Returns the bounding rectangle of this **ccUIGenPoly** shape in the client coordinates specified by the rigid transformation *clientfromShape*.

  Parameters

  clientFromShape The rigid transformation that maps shape coordinates to client coordinates

  Notes

  The client coordinates are defined by a rigid transformation. This type of transformation does not allow you to model scale changes between shape and client coordinates (see *Math Foundations of Transformations* in the *CVL User's Guide*).

clone

```
virtual ccUIGDShape* clone() const;
```

Returns a pointer to a copy of this **ccUIGenPoly** shape.

drawWithXform

```
virtual void drawWithXform (ccUITablet& tablet,
  const ccColor& color, const cc2Xform& xform,
  ccUIShapes::DrawMode m = drawNormal);
```

Draws this **ccUIGenPoly** shape using *tablet* in the color specified by *color.* Before drawing, the shape is mapped to tablet coordinates by using *xform.*

Parameters

tablet	The tablet used to draw this **ccUIGenPoly** shape.
color	The color used to draw this **ccUIGenPoly** shape.
xform	The transformation that maps model coordinates to tablet coordinates.
m	The drawing mode.

updateHandles

```
virtual void updateHandles();
```

Causes all the handles of this **ccUIGenPoly** shape to update themselves.

Notes

Typically, you do not need to call this function since handles are automatically updated by the parent display.

showHandles

```
virtual void showHandles(bool show);
```

If *show* is true, it makes all the handles of this **ccUIGenPoly** shape visible. If false, it makes all the handles of this **ccUIGenPoly** shape invisible.

Parameters

show	Controls whether the handles of this **ccUIGenPoly** shape are visible. If *true* the handles are visible, if *false* they are not.

Notes

Typically, you do not need to call this function since handles are automatically updated by the parent display.

freeHandles

```
virtual void freeHandles();
```

Deletes all the handles of this **ccUIGenPoly** shape.

Notes

Typically, you do not need to call this function since handles are automatically updated by the parent display.

isTouched

```
virtual bool isTouched (const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this **ccUIGenPoly** shape. This function is called automatically for you when you click the mouse on this **ccUIGenPoly** shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw this **ccUIGenPoly** shape
scale	The scale to convert *pt* to pixels

touchDist

```
static float touchDist ();
static void touchDist (float d);
```

- `static float touchDist();`

  Gets the distance from all instances of this object that are still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

  Sets the distance from all instances of this object that are still considered a click on the object. The distance is specified in display coordinates.

  Parameters

d	The distance from the object.

ccUIGenPoly

ccUIGenRect

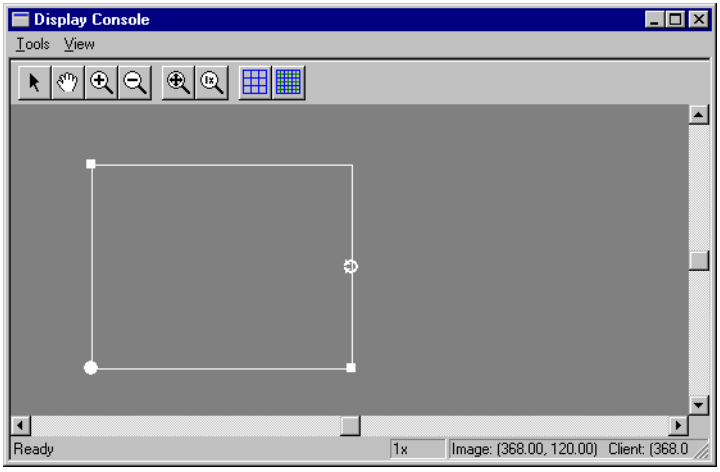
```
#include <ch_cvl/uishapes.h>

class ccUIGenRect : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a generalized rectangle (**ccGenRect**) that you can manipulate. When you add a **ccUIGenRect** object to a **ccDisplay** window a rectangle is displayed that can be moved about, resized, and rotated using your mouse. The following is an example of a generalized rectangle that has been added to a **ccDisplayConsole** window.



ccUIGenRect is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the

ccUIObject and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIGenRect** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUIGenRect** has two resizing handles which both operate in the same way.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the object. The rotation handle is on the right-hand edge of the rectangle.
 Rounded corner handle	Left-click on the rounded corner handle holding the button down, and drag the handle to the object interior to round the corners. Dragging the handle to the object exterior squares the corners. The corner rounding handle is at the lower left-hand corner of the rectangle. This one handle controls corner rounding for all four rectangle corners.

After manipulating a generalized rectangle in a **ccDisplay** window, you will generally want to retrieve the final rectangle state and use it in some way. You can use code similar to the following to retrieve the manipulated rectangle.

```
ccGenRect new_genRect;
new_genRect = ui_genRect->genRect();
```

Where **ui_genRect** is the manipulated **ccUIGenRect** object.

Notes

ccUIGenRect and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIGenRect

```
ccUIGenRect(ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccGenRect**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The rectangle's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Enumerations

eLock

This enumeration lets you specify which handles are not visible when the object is selected. By using these locks, you can prevent a rectangle from being modified along different degrees of freedom.

Value	Meaning
eNoLocks = 0	No locks. All manipulations allowed.
eSizeLock = 0x1	The object cannot be resized.
eRotLock = 0x2	The object cannot be rotated.
eRoundLock = 0x4	The corner radii cannot be changed.
eAspectLock = 0x8	The object keeps the same aspect ratio when resized.
eCenterLock = 0x10	Resizing takes place with respect to the center of the object.
eEllipticLock = 0x20	When changing the size using sizing handles, flat edges are not allowed.

Public Member Functions

genRect

```
void genRect(const ccGenRect& r);

const ccGenRect& genRect() const;
```

- `void genRect(const ccGenRect& r);`

Associates a **ccGenRect** with this manipulation object.

Parameters

r The generalized rectangle to make manipulable.

- `const ccGenRect& genRect() const;`

  Gets the **ccGenRect** associated with this manipulation object.

updateHandles `virtual void updateHandles ();`

Update all of my handles. Called when handles have been repositioned in the display.

showHandles `virtual void showHandles (bool show);`

Make my handles visible/invisible depending on *show*, and the DOF locks. If the handles aren't created yet, create them.

Parameters

show True = show handles. False = do not show handles.

freeHandles `virtual void freeHandles ();`

Deletes all handles.

locks

`c_UInt32 locks() const;`

`void locks(c_UInt32 b);`

- `c_UInt32 locks() const;`

  Returns the lock bits that indicate which resizing operations are not allowed for this object.

- `void locks(c_UInt32 b);`

  Sets the lock bits that indicate the resizing operations that are not allowed for this object.

  Parameters

  b The lock bits. May be any of the following ORed together:

  ccUIGenRect::eNoLocks
  ccUIGenRect::eSizeLock
  ccUIGenRect::eRotLock
  ccUIGenRect::eRoundLock
  ccUIGenRect::eAspectLock
  ccUIGenRect::eCenterLock
  ccUIGenRect::eEllipticLock

  These lock values are described in *Enumerations* on page 3233.

setLocks `void setLocks (c_UInt32 b);`

ORs the specified locks to the current locks.

Parameters

b The locks to OR with the current locks. May be any of the following ORed together:

ccUIGenRect::eNoLocks
ccUIGenRect::eSizeLock
ccUIGenRect::eRotLock
ccUIGenRect::eRoundLock
ccUIGenRect::eAspectLock
ccUIGenRect::eCenterLock
ccUIGenRect::eEllipticLock

These lock values are described in *Enumerations* on page 3233.

clrLocks `void clrLocks (c_UInt32 b);`

Clears the locks specified in *b*. (ANDs the current locks with the complement of *b*)

Parameters

b The locks to clear. May be any of the following ORed together:

ccUIGenRect::eNoLocks
ccUIGenRect::eSizeLock
ccUIGenRect::eRotLock
ccUIGenRect::eRoundLock
ccUIGenRect::eAspectLock
ccUIGenRect::eCenterLock
ccUIGenRect::eEllipticLock

These lock values are described in *Enumerations* on page 3233.

isTouched `virtual bool isTouched (const cc2Vect& pt, double scale);`

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt The position to test in the coordinate system used to draw the shape.

scale The scale to convert *pt* to pixels.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;

virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the center position of the UI shape relative to its parent, in tablet coordinates.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the center position of the UI shape relative to its parent, in tablet coordinates.

  Parameters

pos	The new center position.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal;* draws full genRect *ccUIShapes::drawAbridged;* draws full genRect *ccUIShapes::drawDrag;* draws genRect without rounding

Static Functions

touchDist

```
static float touchDist();
static void touchDist(float d);
```

- `static float touchDist();`

  Gets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

  Sets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

  Parameters

  d The distance from the object.

ccUIGenRect

ccUIIcon

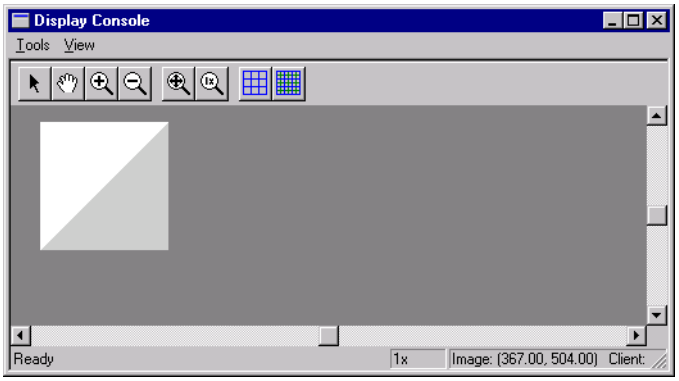
```
#include <ch_cvl/uishapes.h>

class ccUIIcon : public ccUIRLEBuffer;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an icon from a **ccRLEBuffer** object. The icon size cannot be changed and it cannot be rotated. However, you can reposition the icon using your mouse. The displayed icon contains no handles for manipulation. The following is an example of a simple icon that has been added to a **ccDisplayConsole** window.



ccUIIcon is similar to **ccUIRLEBuffer** except **ccUIIcon** does not scale (when you zoom in on a **ccDisplay** window, the icon does not change size) and **ccUIRLEBuffer** does scale. See the **ccUIRLEBuffer** reference page.

ccUIIcon is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the

ccUIObject and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

You can left-click on the icon holding the button down to reposition it in a **ccDisplay** window. Icons cannot be resized and have no manipulation handles.

After repositioning an icon you will generally want to retrieve the final icon state and use it in some way. You can use code similar to the following to retrieve the moved icon.

```
ccRLEBuffer new_rleBuf;
new_rleBuf = ui_icon->icon();
```

Where **ui_icon** is the moved **ccRLEBuffer** object.

Notes

ccUIIcon and any UI shape derived from **ccUIPointShapeBase** uses a coordinate space relative to its parent. When you add an object to a display window its parent is automatically set to one of the coordinate frame objects internal to the display.

Constructors/Destructors

ccUIIcon

```
ccUIIcon (ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccRLEBuffer** used as an icon.

Parameters

parent	The run-length encoded buffer's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

icon

```
void icon(const ccRLEBuffer& buf);

const ccRLEBuffer& icon();
```

- `void icon(const ccRLEBuffer& buf);`

Associates a **ccRLEBuffer** with this manipulation object.

Parameters

buf	The run-length encoded buffer to make manipulable.

Notes

Pixels whose value is **ccColor::passColor()** are not drawn.

- `const ccRLEBuffer& icon();`

  Gets the **ccRLEBuffer** associated with this manipulation object.

touchMap `void touchMap (const TCHAR* map, const ccIPair& org);`

Installs a touch map associated with this icon. The touch map specifies whether a pixel is considered touched when the icon is clicked on. You can use the same patter that you pass to **ccUITablet::makeIcon()** to create the touch map. The plus signs (+) indicate which background areas are considered touched.

Parameters

map	The touch map.
org	The origin relative to the upper left corner of the icon.

isTouched `virtual bool isTouched (const cc2Vect& pt, double scale);`

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

ccUIIcon

ccUILabel

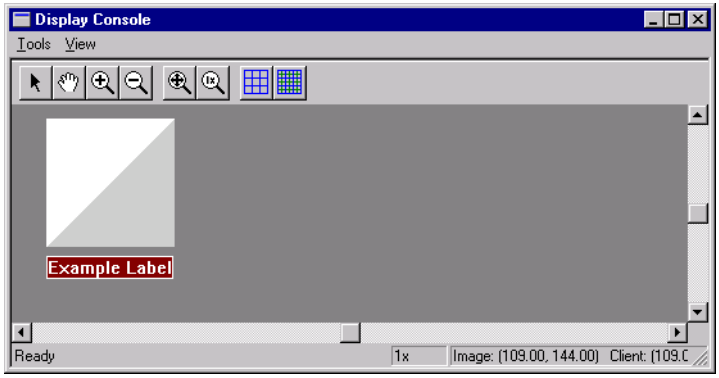
```
#include <ch_cvl/uishapes.h>

class ccUILabel : public ccUIPointShapeBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a label you can use to identify other objects in a display. The label size cannot be changed and it cannot be rotated. However, you can reposition the label using your mouse. The displayed label contains no handles for manipulation. The following is an example of a label that has been added to a **ccDisplayConsole** window. The label is positioned under an icon object to identify it.



ccUILabel is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

You can left-click on the label holding the button down to reposition it in a **ccDisplay** window. Labels cannot be resized and have no manipulation handles.

After repositioning a label in a **ccDisplay** window, you will generally want to retrieve the final label state and use it in some way. You can use code similar to the following to retrieve the moved icon.

```
ccCvlString char_string = ui_label->label();
```

Where **ui_label** is the moved **ccRLEBuffer** object.

Notes

ccUILabel and any UI shape derived from **ccUIPointShapeBase** uses a coordinate space relative to its parent. When you add an object to a display window its parent is automatically set to one of the coordinate frame objects internal to the display.

Constructors/Destructors

ccUILabel

```
ccUILabel(ccUIObject* parent = NULL);
```

Creates a new manipulable object.

Parameters

parent	The label's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

label

```
void label(const ccCvlString& str);
const ccCvlString& label() const;
```

- ```
  void label(const ccCvlString& str);
  ```

  Sets the text of the label.

  Parameters

str	The text of the label.

- ```
  const ccCvlString& label() const;
  ```

  Gets the text of the label.

backColor

```
void backColor(const ccColor& c);
const ccColor& backColor() const;
```

- `void backColor(const ccColor& c);`

  Sets the background color of the label.

  Parameters

  c The color.

- `const ccColor& backColor() const;`

  Gets the background color of the label.

foreColor

```
void foreColor(const ccColor& c);
const ccColor& foreColor() const;
```

- `void foreColor(const ccColor& c);`

  Sets the foreground color (the color of the text) of the label.

  Parameters

  c The color.

  Notes

  The color set via **ccUIShapes::color** is not used during drawing of the label.

- `const ccColor& foreColor() const;`

  Gets the foreground color of the label.

borderColor

```
void borderColor(const ccColor& c);
const ccColor& borderColor() const;
```

- `void borderColor(const ccColor& c);`

  Sets the color of the label border.

  Parameters

  c The label.

- `const ccColor& borderColor() const;`

  Gets the color of the border.

borderWidth

```
void borderWidth(c_Int32 width);
c_UInt32 borderWidth() const;
```

- `void borderWidth(c_Int32 width);`

  Sets the width of the border.

  Parameters

  width The width of the border in pixels.

- `c_UInt32 borderWidth() const;`

  Gets the width of the border.

format

```
void format(const ccUIFormat& f);
const ccUIFormat format() const;
```

- `void format(const ccUIFormat& f);`

  Sets the text format of the label.

  Parameters

  f The format.

- `const ccUIFormat format() const;`

  Gets the label's text format.

isClipped

```
bool isClipped() const;
```

Returns true if some part of the label is clipped in its current position.

isTouched

```
virtual bool isTouched(const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

Protected Member Functions

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

pos_

```
virtual ccPoint pos_ () const;

virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the center position of the UI shape relative to its parent, in tablet coordinates.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the center position of the UI shape relative to its parent, in tablet coordinates.

  Parameters

pos	The new center position.

ccUILabel

ccUILine

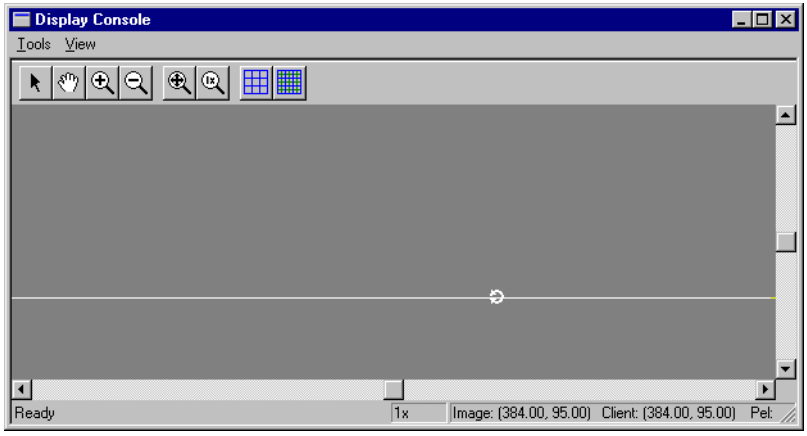
```
#include <ch_cvl/uishapes.h>

class ccUILine : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an infinite-length line (**ccLine**) that you can manipulate. When you add a **ccUILine** object to a **ccDisplay** window the line is displayed and can be moved about and rotated using your mouse. The following is an example of a **ccLine** object that has been added to a **ccDisplayConsole** window.



ccUILine is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the

ccUIObject and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUILine** uses the following handles:

Handle	Description
 Rotation handle	The line rotation handle is a fulcrum around which the line can be rotated. Place your mouse pointer on the rotation handle and hold the left mouse button down. Slide the pointer along the line in either direction and the pull the line up or down to rotate it.

After manipulating a **ccUILine** object in a **ccDisplay** window, you will generally want to retrieve the final line state and use it in some way. You can use code similar to the following to retrieve the manipulated line.

```
ccLine new_line;
new_line = ui_line->line();
```

Where **ui_line** is the manipulated **ccUILine** object.

Notes

ccUILine and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUILine

```
ccUILine(ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccLine**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent	The line's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

line

```
void line(const ccLine& l);
const ccLine& line() const;
```

- `void line(const ccLine& l);`

  Associates a **ccLine** with this manipulation object.

  Parameters

l	The line to make manipulable.

- `const ccLine& line() const;`

  Gets the **ccLine** associated with this manipulation object.

updateHandles

```
virtual void updateHandles ();
```

Update my handles (if created). Called when handles have been repositioned in the display.

showHandles

```
virtual void showHandles (bool show);
```

Make my handles visible/invisible. If the handles aren't created yet, create them.

Parameters

show	True = show handles. False = do not show handles.

freeHandles

```
virtual void freeHandles ();
```

Deletes my handles.

isTouched

```
virtual bool isTouched(const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;

virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns a point through which this line passes, relative to its parent, in tablet coordinates.

  Notes

  pos_() is any point on the line. The **ccVLine** representation has one redundant DOF.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the point through which this line passes, relative to its parent, in tablet coordinates.

  Parameters

  pos The new point.

  Notes

  The new line is parallel to the old line, and passes through the new point *pos*. The rotation handle is placed at *pos*.

dragStop_

```
virtual void dragStop_ (
  const ccIPair& start,
  const ccIPair& stop);
```

An override.

Moves the rotation handle to the ending drag point (*stop*).

Parameters

start Starting drag position in screen coordinates.

stop Ending drag position of the drag in screen coordinates.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag* The draw mode argument is ignored.

Static Functions

touchDist

```
static float touchDist () {return touchDist_;}

static void touchDist (float d) {touchDist_ = d;}
```

- `static float touchDist();`

  Gets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

  Sets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

  Parameters

d	The distance from the object.

ccUILine

ccUILineSeg

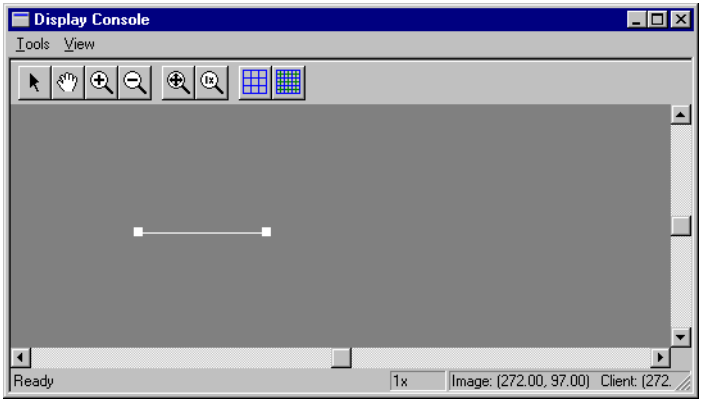
```
#include <ch_cvl/uishapes.h>

class ccUILineSeg : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a line segment (**ccLineSeg**) that you can manipulate. When you add a **ccUILineSeg** object to a **ccDisplay** window the line segment is displayed and can be moved about, resized, and rotated using your mouse. The following is an example of a line segment that has been added to a **ccDisplayConsole** window:



ccUILineSeg is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUILineSeg** uses the following handles:

Handle	Description
 Resize/rotation handle	Left-click on this handle holding the button down and drag the handle to resize the line segment and/or to change its angle. **ccUILineSeg** has two resize/rotation handles which both operate the same way.

After manipulating a **ccUILineSeg** object in a **ccDisplay** window, you will generally want to retrieve the final line segment state and use it in some way. You can use code similar to the following to retrieve the manipulated line segment.

```
ccLineSeg new_lineSeg;
new_lineSeg = ui_lineSeg->lineSeg();
```

Where **ui_lineSeg** is the manipulated **ccUILineSeg** object.

Notes

ccUILineSeg and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUILineSeg

```
ccUILineSeg (ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccLineSeg**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The line segment's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

lineSeg

```
void lineSeg(const ccLineSeg& l);
const ccLineSeg& lineSeg() const;
```

- `void lineSeg (const ccLineSeg&);`

  Associates a **ccLineSeg** with this manipulation object.

  Parameters

l	The line segment to make manipulable.

- `const ccLineSeg& lineSeg() const;`

  Gets the **ccLineSeg** associated with this manipulation object.

updateHandles

```
virtual void updateHandles ();
```

Update my handles (if created). Called when handles have been repositioned in the display.

showHandles

```
virtual void showHandles (bool show);
```

Make my handles visible/invisible. If the handles aren't created yet, create them.

Parameters

show	True = show handles. False = do not show handles.

freeHandles

```
virtual void freeHandles ();
```

Deletes the handles.

isTouched

```
virtual bool isTouched(const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;

virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the position of the line segment relative to its parent, in tablet coordinates. The position is the line segment center.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the center position of the line segment relative to its parent, in tablet coordinates.

  Parameters

pos	The new center position.

  Notes

  The new line segment passes through *pos* and is parallel to the old line segment. *pos* is the new line segment center.

dragStop_

```
virtual void dragStop_ (
  const ccIPair& start,
  const ccIPair& stop);
```

An override.

Moves the line segment to the ending drag point (*stop*).

Parameters

start	Starting drag position in screen coordinates.
stop	Ending drag position of the drag in screen coordinates.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the line segment in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag* The draw mode argument is ignored.

Static Functions

touchDist

```
static float touchDist();
static void touchDist(float d);
```

- `static float touchDist();`

  Gets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

  Sets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

  Parameters

d	The distance from the object.

ccUILineSeg

ccUIManShape

```
#include <ch_cvl/uishapes.h>

class ccUIManShape : public ccUIShapes;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This is an abstract class that is used as the base class for all the manipulable shape classes. This class provides the functions for making a shape's handles visible or invisible and whether the handles should be visible when the shape is selected.

Constructors/Destructors

Since this is a pure virtual class, you cannot create an instance of it.

Public Member Functions

manipulable

```
void manipulable(bool m);

bool manipulable() const;
```

- `void manipulable(bool m);`

  Sets whether this shape is manipulable.

  Parameters

m	True if this shape's handles should be visible when the shape is selected, false otherwise.

- `bool manipulable() const;`

  Gets whether this shape is manipulable.

updateHandles

```
virtual void updateHandles () = 0;
```

Update all of my handles. Called when handles have been repositioned in the display.

showHandles `virtual void showHandles (bool show) = 0;`

Make my handles visible/invisible. If the handles aren't created yet, create them.

Parameters

show	True = show handles. False = do not show handles.

freeHandles `virtual void freeHandles () = 0;`

Deletes all handles.

Protected Member Functions

select `virtual void select (States s, bool parentChanged);`

An override.

The UI Framework calls **ccUIObject::select()** when the object's state changes to *selectedState* (the object becomes selected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

This override function calls **showHandles()** to make the handles visible.

Parameters

s	The state change that caused the object to become selected. Must be one of the **ccUIObject::States** enums.
parentChanged	True if the select was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

deselect `virtual void deselect (States s, bool parentChanged);`

An override.

The UI Framework calls **ccUIObject::deselect()** when the object's state changes from *selectedState* (the object becomes deselected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

This override function calls **showHandles()** to make the handles invisible.

Parameters

s	The state change that caused the object to become selected. Must be one of the **ccUIObject::States** enums.
parentChanged	True if the select was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

ccUIManShape

ccUIObject

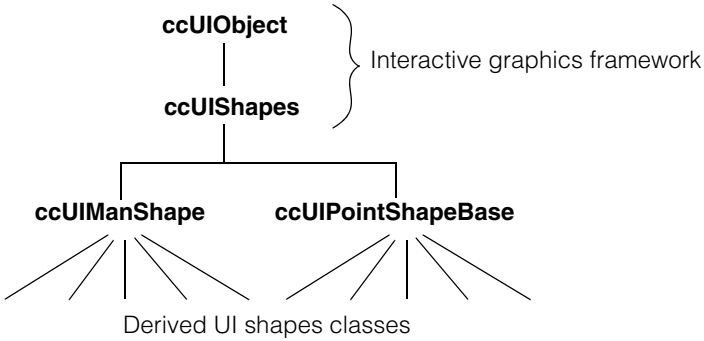
```
#include <ch_cvl/uifrmwrk.h>

class cmImport_cogdisp ccUIObject;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

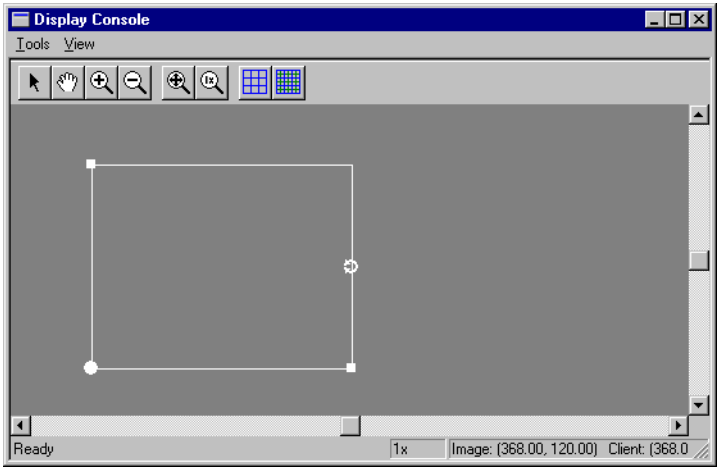
ccUIObject and **ccUIShapes** are base classes for the interactive graphics classes, also called UI shapes classes. These base classes make up the interactive graphics framework. See the following class derivation hierarchy.



Interactive graphics classes use the graphics classes defined in *shapes.h* and include fundamental shapes such as the line, rectangle, circle, ellipse, and others. These interactive classes provide an environment where shapes can be manipulated either graphically using a mouse and a **ccDisplay** window, or with your program using member functions from the framework base classes.

For example, an interactive generalized rectangle (**ccUIGenRect**) class derived from **ccUIObject** and **ccUIShapes** creates a generalized rectangle (**ccGenRect**) that you can manipulate. When you add a **ccUIGenRect** object to a **ccDisplay** window a

rectangle is displayed that can be moved about, resized, and rotated using your mouse or through program control. The following is an example of a generalized rectangle that has been added to a **ccDisplayConsole** window.



Interactive graphic objects displayed in a **ccDisplay** window have the following common graphic features:

Feature	Description
Selecting an object	To manipulate an object you must first select it by left-clicking your mouse pointer on the object outline. Your mouse pointer changes to a cross when you are close enough to select the object. Once selected the object changes to the color set by the last call to **selectColor()**. The default color is yellow. Selected objects are displayed with handles you use to manipulate the object. The handles available with UI shapes objects are described below.
Deselecting an object	You deselect an object by selecting another object or by left-clicking your mouse in free space anywhere in the window. Deselected objects return to their original color which you can set by calling **ccUIShapes::props()**. The default color is cyan.
Moving a selected object	To move a selected object, left-click on the object outline holding the mouse button down, and drag the object to the new location.

Feature	Description
Handles	Handles appear as magenta icons on selected objects. You cannot change the handle color. The following handles are used, although every handle is not available on all shapes.
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the object.
 Rounded corner handle	Left-click on the rounded corner handle holding the button down, and drag the handle to the object interior to round the corners. Dragging the handle to the object exterior squares the corners.
 Skew handle	Left-click on the skew handle holding the button down, and drag the handle to change the object's skew.

Member Function Summary

ccUIObject is a large and complex class. Use the following summaries as a guide to the class capabilities, and as a learning tool. The class member functions can be divided into the following functional groups:

Framework-only	Relationships	States	Information
shapes()^{V}	**parent()^{V}**	**condVisible()**	**wholsTouched()^{V}**
frame()	**frontSib()**	**visible()**	**opNew()**
shapesFrame()	**backSib()**	**condEnabled()**	**root()**
rootObject()	**closerSib()^{V}**	**enabled()**	**rootMutex()**
tablet()^{V}	**fartherSib()^{V}**	**condSelected()**	**orphanMutex()**
	frontKid()^{V}	**selected()**	**uiMutex()**
	backKid()^{V}	**condVisAndEnab()**	**key()**
	numKids()^{V}	**condVisEnabAndSel()**	**isValid()**

(**V**) Virtual function.

You should not call Framework-only classes in your program. *Relationships*, *States*, and *Information* refer to the object from which they are called.

Attributes	Graphics commands	Protected virtual functions
clickable()		**show()**
draggable()		**hide()**
dragging()		**enable()**
dontMove()		**disable()**
keepSel()		**select()**
dwell()		**deselect()**
multiSelectable()	**mouseDown()**	**mouseDown_()**
multiSelected()	**mouseUp()**	**mouseUp_()**
multiDraggable()	**mouseEnter()**	**mouseEnter_()**
userSelect()	**idleMouseEnter()**	**idleMouseEnter_()**
rightButtonMode()	**mouseLeave()**	**mouseLeave_()**
autoDelete()	**mouseMove()**	**mouseMove_()**
mark()	**click()**	**click_()**
selectColor()	**dblClick()**	**dblClick_()**
deselColor()	**dragStart()**	**dragStart_()**
dragColor()	**dragStop()**	**dragStop_()**
curSelColor()	**dragAnimate()**	**dragAnimate_()**
lightColor()	**mouseMiddle()**	**mouseMiddle_()**
shadowColor()	**mouseRight()**	**mouseRight_()**
testColor()	**front()**	**front_()**
faceColor()	**back()**	**back_()**
drawLayer()	**keyboard()**	**keyboard_()**

Attributes allow you to tailor objects for your application. *Graphics commands* are functions called by the UI framework in response to mouse and keyboard events, and can be also be called in your programs to simulate these events. *Protected virtual functions* perform no operations here, but are place holders for derived class overrides. Most of these virtual functions are called by a corresponding graphics command. For example, **click()** calls **click_()** which performs no operation unless overridden in a derived class.

For additional information about interactive graphics classes, see the *Displaying Graphics* chapter of the *CVL User's Guide*.

Constructors/Destructors

This is an abstract class. Only classes derived from this class use the constructors and destructors.

ccUIObject `protected: ccUIObject();`

Default constructor. **opNew()** and **autoDelete()** are true if created with the new operator. All other properties are set false.

~ccUIObject `virtual ~ccUIObject();`

Destructor. Tells the event processor that the UI object is being deleted and removes it from the registry of all objects.

Notes

Other obvious actions, such as notifying parents/kids and making the UI object invisible, must be done by derived objects, as the UI object has now been destroyed to the point where it can no longer find its parents, kids, or tablet.

Enumerations

RightButtonModes

This enumeration defines the actions to be taken when the user clicks the right mouse button on a UI object.

Value	Meaning
eIgnoreRB	Ignore right-mouse clicks.
eClientRB	Call **mouseRight()** so a user-derived class can handle right-mouse click events.
eBackRB	Move the UI object in back of other objects.
eShiftRB	Simulate Shift + left-mouse click behavior.

States

This enumeration specifies the three possible states of a **ccUIObject**.

Value	Meaning
visibleState	The object is displayed on the screen and is visible unless covered by another object. The following conditions specify this state: **visible()** = true **condVisible()** = true Parent **visible()** = true
enabledState	The object is enabled and responds to mouse clicks. The following conditions specify this state: **enabled()** = true **condVisible()** = true **condEnabled()** = true Parent **enabled()** = true
selectedState	The object is displayed using its selected color, showing handles (if any). The following conditions specify this state: **selected()** = true **condVisible()** = true **condEnabled()** = true **condSelected()** = true Parent **selected()** = true

Public Member Functions

shapes

```
virtual ccUIShapes* shapes(bool canFail = false);
```

RTTI framework operation providing base-to-derived type conversion for framework classes. Returns the *this* pointer if the object is one of the **ccUIShapes**, otherwise 0. Relatively lightweight as compared to *dynamic_cast<>*.

Parameters

canFail — Specifies whether or not the operation can fail. False asserts that the conversion can be done and causes a valid pointer always to be returned. A non-zero value causes NULL to be returned if the conversion cannot be done.

Notes

This function is for Cognex internal use only.

frame `virtual ccUIFrame* frame(bool canFail = false);`

RTTI framework operation providing base-to-derived type conversion for framework classes. Returns the *this* pointer if the object is a **ccUIFrame**, otherwise 0. Relatively lightweight as compared to *dynamic_cast<>*.

Parameters

canFail — False means the conversion cannot fail and a valid pointer is always returned. True means the operation can fail and NULL is returned if the conversion cannot be done.

Notes

This function is for Cognex internal use only.

shapesFrame `virtual ccUIShapesFrame* shapesFrame(bool canFail = false);`

RTTI framework operation providing base-to-derived type conversion for framework classes. Returns the *this* pointer if the object is a **ccUIShapesFrame**, otherwise 0. Relatively lightweight as compared to *dynamic_cast<>*.

Parameters

canFail — Specifies whether or not the operation can fail. False asserts that the conversion can be done and causes a valid pointer always to be returned. A non-zero value causes NULL to be returned if the conversion cannot be done.

Notes

This function is for Cognex internal use only.

rootObject `virtual ccUIRootObject* rootObject(bool canFail = false);`

RTTI framework operation providing base-to-derived type conversion for framework classes. Returns the *this* pointer if the object is a **ccUIRootObject,** otherwise 0. Relatively lightweight as compared to *dynamic_cast<>*.

Parameters

canFail — Specifies whether or not the operation can fail. False asserts that the conversion can be done and causes a valid pointer always to be returned. A non-zero value causes NULL to be returned if the conversion cannot be done.

Notes

This function is for Cognex internal use only.

parent

```
virtual ccUIObject* parent() = 0;
```

Gets the parent (runtime owner) of the UI object. This pure virtual function is implemented in the derived class **ccUIShapes**.

closerSib

```
virtual ccUIObject* closerSib() = 0;
```

Returns the sibling object added to the parent object just after this object. If there is none, it returns 0.

This pure virtual function is implemented in the derived class **ccUIShapes**.

The following is an example of code that uses this function to navigate the runtime hierarchy of a UI object:

```
void dumpObjects(ccUIObject* obj, int level = 0) {
  cogOut << ccCvlString(level*3, ' ') << typeid(*obj.name() <<
    cmStd endl;
  for(ccUIObject* p = obj->backKid(); p; p = p->closerSib())
    dumpObjects(p, level + 1);
}
```

Calling:

```
  dumpObject(d.root());
```

on a display console that was created with:

```
  ccDisplayConsole d(...);
```

will yield the following output, showing the runtime object hierarchy of the display console:

```
class ccUIWin32Root
  class ccDisplayConsole
    class ccUINestedCoordFrame
    class ccUINestedCoordFrame
    class ccUIShapesFrame
  class UIFrameSizer
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
    class ccUIFrameSizeHandle
```

fartherSib

```
virtual ccUIObject* fartherSib() = 0;
```

Returns the sibling object added to the parent object just before this object. If there is none, it returns 0.

This pure virtual function is implemented in the derived class **ccUIShapes**.

frontKid

```
virtual ccUIObject* frontKid() = 0;
```

Returns the child object of this shape that was last added. If there is none, it returns 0.

This pure virtual operation is implemented in the derived class **ccUIShapes**.

backKid

```
virtual ccUIObject* backKid() = 0;
```

Returns the child object of this shape that was first added. If there is none, it returns 0.

This pure virtual operation is implemented in the derived class **ccUIShapes**. See **closerSib** for sample code that uses this function to navigate the runtime object hierarchy of a UI object.

frontSib

```
ccUIObject* frontSib();
```

Returns this object's parent's last added child. If this object was the last added, it returns itself.

backSib

```
ccUIObject* backSib();
```

Returns this object's parent's first added child. If this object was the first added, it returns itself.

numKids

```
virtual c_UInt32 numKids() = 0;
```

Gets the number of kids of the UI object. This pure virtual operation is implemented in the derived class **ccUIShapes**.

tablet

```
virtual ccUITablet& tablet() = 0;
```

Gets the tablet in which the UI object is to be drawn. The **draw()** method in a derived UI shape class draws into this tablet.

This pure virtual function is implemented in the derived class **ccUIShapes**.

Notes

The tablet and scope coordinates of the returned tablet are often referred to as the tablet/scope coordinates of this **ccUIObject**. You can use them for the mouse interface methods.

root

```
ccUIRootObject* root();
const ccUIRootObject* root() const;
```

- `ccUIRootObject* root();`

  Gets the root object of the UI object, if it has one.

- `const ccUIRootObject* root() const;`

  Gets a const pointer to the root object of the UI object, if it has one.

rootMutex

```
ccMutex& rootMutex();
```

Gets the root mutex of the UI object, if it has one. If the UI object does not have a root mutex, the orphan mutex is returned. If the UI object is itself a root, then its own mutex is returned.

Root mutex is useful for thread synchronization.

drawLayer

```
void drawLayer(ccUITablet::Layers lyr);
ccUITablet::Layers drawLayer() const;
```

- `void drawLayer(ccUITablet::Layers lyr);`

  Specifies the layer into which this object should be drawn. The derived classes, which do the actual drawing, may choose to ignore this value.

  The default is **ccUITablet::eImageLayer**. The draw layer for a **ccUIObject** must be specified before calling **ccDisplay::addShape()**.

  Parameters

  lyr — Layer into which the UI object should be drawn. Must be one of:

  ccUITablet::eImageLayer (default)
  ccUITablet::eOverlayLayer

Notes

Not all layers are supported on all platforms. You can use the static member function **ccUITablet::overlaySupported()** to determine whether your platform supports overlays.

All interactive shapes are drawn to the image layer by default. If you want to draw the shape on a different layer (for example, the overlay layer), you can set the drawing layer using **ccUIObject::drawLayer(lyr)**. Cognex recommends changing the drawing layer before adding the shape to the display. Changing the drawing layer once a shape has been added to the display will result in lower performance when the graphics are re-rendered.

When assigning a parent to interactive shapes using **ccUIShapes::parent(newParent)**, all children must be on the same drawing layer as the parent. Cognex recommends setting the parent of an interactive shape before adding the shape to the display. Changing a shape's parent once the shape has been added to the display will result in lower performance when the graphics are re-rendered.

For optimal rendering performance, do not change the drawing layer or parent of an interactive shape after adding the shape to the display console.

- `ccUITablet::Layers drawLayer() const;`

Returns the layer into which the UI object should be drawn.

key `c_UInt32 key() const;`

Returns a key that uniquely identifies this object.

Notes

Do not use the address of the UI object (the *this* pointer) as an identifier because it can become non-unique over time. An object could be deleted and a new one constructed at the same address.

curSelColor `ccColor curSelColor() const;`

Returns **selectColor()** if the object is selected. Returns **deselColor()** if the object is deselected.

deselColor `ccColor deselColor() const;`

Returns the color used to indicate that an object is deselected. This color is set in the derived class function **ccUIShapes::props()**.

lightColor `ccColor lightColor() const;`

Returns a light gray color.

Notes

Use **lightColor()**, **shadowColor()**, **testColor()**, and **faceColor()** instead of coding the actual color. Then, if these colors are changed in the future, you will not have to change your application code.

shadowColor

```
ccColor shadowColor() const;
```

Returns a dark gray color, generally used for shadows.

textColor

```
ccColor textColor() const;
```

Returns the color black, generally used for text.

faceColor

```
ccColor faceColor() const;
```

Returns a medium gray color, generally used for faces.

---

condVisible

```
bool condVisible() const;
bool condVisible(bool vis);
```

- `bool condVisible() const;`

  Returns true if the UI object itself is visible, false otherwise. If the UI object's parent is not visible, you will not see the UI object on the screen even if **condVisible()** is true. The parent of a shape is usually the **ccDisplay** window in which it is drawn and is usually visible.

- `bool condVisible(bool vis);`

  Sets whether the UI object itself is visible. If the UI object's parent is not visible, you will not see the UI object on the screen even if **condVisible()** is true.

  Parameters

vis	If true, the UI object is visible when the parent is visible. In CVL, the parent of a shape is usually the **ccDisplay** window in which it is drawn and is usually visible. So, setting *vis* to true normally makes a shape visible.

visible

```
bool visible();
```

Returns true if the UI object is visible, false otherwise. The UI object is visible if **condVisible()** returns true and its parent's **condVisible()** also returns true.

condEnabled

```
bool condEnabled() const;
bool condEnabled(bool enab);
```

- `bool condEnabled() const;`

  Returns true if the UI object itself is enabled, false otherwise. If the UI object's parent is not enabled, the UI object will not be able to respond to mouse events even if **condEnabled()** is true. The parent of a shape is usually the **ccDisplay** window in which it is drawn and is usually enabled.

  Notes

  A conditionally enabled object does not necessarily respond to mouse-click events. See **clickable()** on page 3283.

- `bool condEnabled (bool enab);`

  Sets whether the UI object itself is enabled. If the UI object's parent is not enabled, the UI object will not be able to respond to mouse events, even if **condEnabled()** is true.

  Parameters

enab	If true, the UI object can receive mouse events.

  Notes

  A conditionally enabled object does not necessarily respond to mouse-click events. See **clickable()** on page 3283.

enabled

```
bool enabled();
```

Returns true if the UI object is enabled (can receive mouse events), false otherwise. The UI object is enabled if **condEnabled()** returns true and its parent's **condEnabled()** also returns true.

condSelected

```
bool condSelected() const;
bool condSelected (bool sel, bool kidsToo = false);
```

- `bool condSelected() const;`

  Returns true if the UI object itself is selected, false otherwise. If the UI object's parent is not selected, the UI object will not appear selected in the display even if **condSelected()** is true. The parent of a shape is usually the **ccDisplay** window in which it is drawn unless the shape is part of a parent/child relationship.

- `bool condSelected(bool sel, bool kidsToo=false);`

  Sets whether the UI object itself is selected. If the UI object's parent is not selected, the UI object will not appear selected in the display even if **condSelected()** is true.

  If **keepSel()** is true, the *sel* parameter is ignored in this call. This means you cannot deselect an object when **keepSel()** is true.

  Parameters

sel	If true, the UI object is selected when its parent is selected.
kidsToo	If true, the UI object's children are also selected.

selected

```
bool selected();
```

Returns true if the UI object is selected, false otherwise. The UI object is selected if **condSelected()** returns true and its parent's **condSelected()** also returns true.

condVisAndEnab

```
bool condVisAndEnab() const;
```

Returns true if the UI object is both visible and enabled (**visible()** = true, **enabled()** = true). Returns false otherwise.

condVisEnabAndSel

```
bool condVisEnabAndSel();
```

Returns true if the UI object is visible, enabled, and selected (**visible()** = true, **enabled()** = true, **selected()** = true). Returns false otherwise.

multiSelectable

```
bool multiSelectable() const;

void multiSelectable(bool sel);
```

Specifies whether this object can be simultaneously selected along with one or more other objects. Multi-selected objects can be dragged around together retaining their distance and orientation relative to one another. You multi-select by left-clicking each object while holding down the shift key.

You can prevent an object from being multi-selected by setting this flag false.

Notes

Multi-selection only works on objects that are siblings of one another. In complex parent/child applications check carefully for this condition.

- `bool multiSelectable() const;`

  Returns true if the UI object is set to be multi-selectable. Returns false otherwise.

  Example

  To determine whether the UI object and another are simultaneously selected, use the following code:

```
  // other is a pointer to the other object
if (selected() && multiSelectable()
     && other->selected() && other->multiSelectable()) {
   ...
}
```

- `void multiSelectable(bool sel);`

  Sets the UI object's multi-select flag.

  Parameters

  sel — The new multi-select flag, true of false.

multiSelected `bool multiSelected(const ccUIObject* sib) const;`

Returns true if the specified sibling's multi-select flag is true, meaning it can be multi-selected. Returns false otherwise.

Parameters

sib — The sibling UI object you wish to test. Can be either a sibling or the UI object itself. This method always returns true if *sib* is the UI object itself.

Notes

We do not recommended that you use the **multiSelected()** method in your own programs because its name is not entirely consistent with its implementation. Instead use **multiSelectable()** (see the example code provided with that method).

multiDragable `bool multiDragable();`

Returns true if the UI object and all of its multi-selected siblings can be dragged. Returns false otherwise.

Notes

The **multiDragable()** method is the preferred way to determine if an object can be dragged. The **draggable()** method, on the other hand, determines whether an object is conditionally draggable (that is, draggable if other conditions also hold).

userSelect `void userSelect(c_Uint32 shift);`

Specifies how to implement click-select behavior in combination with the Shift key.

Parameters

shift Specifies the meaning of the Shift key in combination with a mouse button click.

Value	Effect
shift = -1	If **enabled()** == true (requires **condEnabled()** == true and **condVisible()** == true) do the following: • Deselect all sibling objects (using **condSelected(**false**)**). • Select me (using **condSelected(**true**)**. • If **dontMove()** == false, call **front()**. • Recurse up the object hierarchy using **parent()** as the next object. -or- If **enabled()** == false, do nothing on this object but still recurse up the object hierarchy using **parent()** as the next object.

Value	Effect
shift = 0	If **enabled()** == true (requires **condEnabled()** == true and **condVisible()** == true) do the following: • If I am not currently selected, deselect all sibling objects (using **condSelected(**false**)**). • Select me (using **condSelected(**true**)**. • If **dontMove()** == false, call **front()**. • Recurse up the object hierarchy using **parent()** as the next object. -or- If **enabled()** == false, do nothing on this object but still recurse up the object hierarchy using **parent()** as the next object.

shift = 1	if **enabled()** == true and **multiSelectable()** = true, do the following:
	• If I am about to become selected, deselect all my non-multiSelectable siblings.
	• Toggle my conditional-selected state.
	• Recurse up the object hierarchy using **parent()** as the next object.
	-or-
	if **enabled()** == true and **multiSelectable()** = false, do the following:
	• If I am not currently selected, deselect all my non-multiSelectable siblings.
	• Select me (using **condSelected(**true**)**.
	• If **dontMove()** == false, call **front()**.
	• Recurse up the object hierarchy using **parent()** as the next object.

Notes

1. The difference between using a *shift* value of 0 or -1 affects the deselection of other objects. A shift value of 0 will only deselect other objects if the current object is not selected. This choice can be used to mimic the behavior caused by clicking an object with the mouse, or to mimic shift_key-click multi-selectability with the mouse. To mimic a single action mouse-click on this object, call **userSelect(**0**)**. To mimic shift_key-click multi-selectability action, call **userSelect(**1**)**.
2. This function differs from **condSelected(**bool, bool**)** in that it changes the **condSelected()** state of many more objects, including sibling and parent objects. If you wish to modify only the conditional state of this object, use **condSelected(**bool, bool**)**.
3. For this object or any parent object above in the object hierarchy that is visited from this function, **enabled()** must be true for any action to occur. For this object or any parent object above in the object hierarchy with **enabled()** == false, no action is taken. Sibling objects and children objects of this object do not consider the **enabled()** state when changing the selected state.
4. The value of **keepSel()** controls whether an object can be deselected. If **keepSel()** == true then that object will not be deselected by this function call. This function may change the selected state of an object and as such may call **select_()**, **deselect_()**, and/or **front_()**.

whoIsTouched

```
virtual ccUIObject* whoIsTouched(const ccIPair& p);
```

Returns a pointer to the front-most, visible UI object touched by the point *p* that is either this object or one of its descendants. Returns NULL if the point touches neither this UI object nor any of its descendants.

Parameters

p	Position to be checked for proximity to UI objects.

Notes

The point is specified in the scope coordinates of the UI object's parent. For the root object, these are screen coordinates. For shapes, they are the same as the UI object's tablet coordinates.

Notes

Only framework classes should override this method. Override **ccUIFrame::isTouched()** and **ccUIShapes::isTouched()** in classes you derive to specify how particular objects are touched.

front

```
void front();
```

Called by the UI framework when the object is moved to the front. (For example, when it is selected). This function calls **front_()** which can be overridden in a user-derived class.

back

```
void back();
```

Called by the UI framework when the object is moved to the back. This function calls **back_()** which can be overridden in a user-derived class.

clickable

```
bool clickable() const;
void clickable(bool click);
```

- `bool clickable() const;`

  Returns true if the UI object can accept mouse clicks. Returns false otherwise.

- `void clickable(bool click);`

  Sets the UI object's clickable flag, which determines whether or not the UI object can accept mouse clicks.

  Parameters

  click — If true, the UI object is clickable.

draggable

```
bool draggable() const;
void draggable(bool drag);
```

- `bool draggable() const;`

  Returns true if the UI object can be dragged with the mouse. Returns false otherwise. See also **multiDragable()**.

- `void draggable(bool drag);`

  Sets the UI object's draggable flag, which determines whether or not the UI object can be dragged with the mouse.

  Parameters

  drag — If true, the UI object is draggable.

dragging

```
bool dragging() const;
void dragging(bool);
```

- `bool dragging() const;`

  Returns true if the UI object is currently being dragged with the mouse. Returns false otherwise.

- `void dragging(bool drag);`

  Sets the UI object's dragging flag. This is an internal UI framework function. Do not call this function from your program.

  Parameters

  drag — If true, the UI object is currently being dragged.

  Notes

  In your program you can test, but should not set, whether the UI object is being dragged.

dontMove

```
bool dontMove() const;
void dontMove(bool move);
```

- `bool dontMove() const;`

  Returns true if the UI object should not move in front of other objects when it is selected. Returns false otherwise.

- `void dontMove(bool noMove);`

  Sets the UI object's dontMove flag, which determines whether or not the UI object should move to the front when it is selected. Set this flag to true for UI objects that should maintain a fixed screen order that is independent of user mouse clicks.

  Parameters

  noMove — If true, the UI object will not move in front of other objects when selected.

opNew

```
bool opNew() const;
```

Returns true if the UI object was created with **new()** and therefore requires a separate **delete()** to destroy it.

Notes

The **opNew()** method indicates whether the UI object was created with the operator **new()**, but not whether it is currently on the heap. An object can also be on the heap if it was created not directly with the operator **new()** but as a member of another object that was. Such a component object is destroyed when the composite object that contains it is destroyed, and does not require a separate **delete()**.

autoDelete

```
bool autoDelete() const;

void autoDelete(bool autoDel);
```

UI objects record whether they have been created on the heap or the stack so they can be automatically deleted. A **ccUIObject** can do this because it overrides the operator **new()**. This allows you to add a number of shapes to the display without having to remember to clean up after them. Some objects never use **autoDelete()**. For example, **ccWin32Display** disables **autoDelete()** in its constructor.

Notes

Since **ccDisplay** automatically manages UI shapes when added, you should not create UI shapes on the heap. Use stack objects only if you call **autoDelete(***false***)**.

- `bool autoDelete() const;`

  Returns true if the UI object should be deleted when its parent is deleted. Returns false otherwise.

- `void autoDelete(bool autoDel);`

  Sets the UI object's autoDelete flag.

  Parameters

  autoDel If true, the UI object is deleted when its parent is deleted.

  Notes

  The UI object's **opNew()** flag must be set to true before the autoDelete flag can be set to true. The autoDelete flag is set to true by default if the object is created with **new()**, otherwise the flag is set to false.

  Notes

  Calling *display->autoDelete(true)* can crash the display system even if the display was allocated on heap. This is true because the **ccWin32Display** constructor deletes the root **ccUIObject**, which in turn deletes all of its children. Since **ccWin32Display** is a child of **ccUIObject**, if **autoDelete()** is true the display will be deleted twice.

mark

```
bool mark() const;
void mark(bool mflag);
```

- `bool mark() const;`

  Returns true if the UI object is marked. Returns false otherwise.

- `void mark(bool);`

  Sets the UI object's *marked* flag.

  Parameters

  mflag True marks the UI object, false unmarks it.

  Notes

  Clients can mark a UI object for any purpose useful in their application.

keepSel

```
bool keepSel() const;
void keepSel(bool keep);
```

This function is useful when treating multiple UI shapes as a single entity. For example, each shape can be set to remain selected so that all the shapes can be dragged together.

When set to true, **condSelected(false)** calls are ignored.

If the parent becomes deselected **keepSel()** is set to false and **condSelected()** is set to false.

- `bool keepSel() const;`

  Returns true if the UI object cannot be deselected. Returns false otherwise.

- `void keepSel(bool keep);`

  Sets the UI object's keepSel flag, which determines whether or not the object can be deselected. If true, the UI object cannot be deselected and its **condSelected()** flag is also set to true.

  Parameters

  keep If true the UI object cannot be deselected.

dwell

```
bool dwell() const;

void dwell(bool dflag);
```

When **dwell()** is false, each **mouseDown()** - **mouseUp()** pair generates a **click()** event. In addition, when **dwell()** is true, multiple **click()** events are generated at a fixed rate as long as the left mouse button is held down.

- `bool dwell() const;`

  Returns true if the UI object can generate multiple click events while the user holds the left mouse button down.

- `void dwell(bool dflag);`

  Sets the UI object's dwell flag, which determines whether or not it can generate multiple click events while the user holds down the left mouse button.

  Parameters

  dflag — New dwell flag, true or false.

rightButtonMode

```
RightButtonModes rightButtonMode();

void rightButtonMode(RightButtonModes m);
```

- `RightButtonModes rightButtonMode();`

  Returns the UI object's right button mode, which specifies how the object handles right mouse button click events.

- `void rightButtonMode(RightButtonModes m);`

  Sets the UI object's right button mode.

  Parameters

  m — Right mouse button mode. Must be one of the **RightButtonModes** enums (*eIgnoreRB, eClientRB, eBackRB, eShiftRB*).

  See the **RightButtonModes** enum on page 3269.

mouseMiddle

```
void mouseMiddle(const ccIPair& p, MouseAction event,
                 c_UInt32 keys);
```

This method is invoked by the UI framework for a mouse middle button event when the mouse is pointing to this object. **mouseMiddle()** calls **mouseMiddle_()**, a function you must override in your own derived UI class if you wish to implement a mouse middle button response.

Parameters

p	Last position of the mouse in screen coordinates.
event	Type of mouse event that caused this call. Must be one of the **ccMouseEvent::Event** enums (*eNone, eMovement, eDownClick, eDoubleClick, eUpClick, eDwell*).
keys	Modifier keys depressed when the mouse action occurred. Must be one of the **ccMouseEvent::Key** enums (*eNoKey*, *eAlt*, *eControl*, *eShift*).

mouseRight

```
void mouseRight(const ccIPair& p, MouseAction event,
                c_UInt32 keys);
```

This method is invoked by the UI framework for a mouse right button event when the UI object is in the *eClientRB* mode (**rightButtonMode()** = *eClientRB*), and the mouse is pointing to this object. **mouseRight()** calls **mouseRight_()**, a function you must override in your own derived UI class if you wish to implement a mouse right button response.

Parameters

p	Last position of the mouse in screen coordinates.
event	Type of mouse event that caused this call. Must be one of the **ccMouseEvent::Event** enums (*eNone, eMovement, eDownClick, eDoubleClick, eUpClick, eDwell*).
keys	Modifier keys depressed when the mouse action occurred. Must be one of the **ccMouseEvent::Key** enums (*eNoKey*, *eAlt*, *eControl*, *eShift*).

mouseDown

```
void mouseDown();
```

The UI Framework calls **mouseDown()** when the left mouse button is pressed down while pointing to this object (the object must be **clickable()**). **mouseDown()** then calls the protected function **mouseDown_()**. If **mouseDown_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseDown_()** is overridden in a user-derived class, the override function is executed.

mouseUp `void mouseUp();`

The UI Framework calls **mouseUp()** following a mouseDown when the left mouse button is released, or when the mouse pointer is moved off the object. **mouseUp()** then calls the protected function **mouseUp_()**. If **mouseUp_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseUp_()** is overridden in a user-derived class, the override function is executed.

mouseEnter `void mouseEnter();`

The UI Framework calls **mouseEnter()** when the mouse enters the UI object's screen space with a left mouse button held down. **mouseEnter()** then calls the protected virtual function **mouseEnter_()** which is overridden in **ccUIShapes::mouseEnter_()** where the actual implementation is done.

idleMouseEnter `void idleMouseEnter();`

The UI Framework calls **idleMouseEnter()** when the mouse enters the UI object's screen space with the left mouse button up. **idleMouseEnter()** then calls the protected function **idleMouseEnter_()** which is overridden in **ccUIShapes::idleMouseEnter_()** where the actual implementation is done

mouseLeave `void mouseLeave();`

The UI Framework calls **mouseLeave()** when the mouse is moved off the UI object, or also, if an enabled object was entered and it then becomes disabled. **mouseLeave()** then calls the protected function **mouseLeave_()** which is overridden in **ccUIShapes::mouseLeave_()** where the actual implementation is done

mouseMove `void mouseMove(const ccIPair& screenPos);`

The UI Framework calls **mouseMove()** repeatedly when the mouse is moving while pointing to this object. (For example, while resizing the object). Each call includes the current mouse position in screen coordinates. **mouseMove()** then calls the protected function **mouseMove_()**. If **mouseMove_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseMove_()** is overridden in a user-derived class, the override function is executed.

Parameters

screenPos	The current mouse position in screen coordinates.

click

```
void click();
```

The UI Framework calls **click()** following a **mouseDown()** - **mouseUp()** sequence. Also, following a **mouseDown()**, if **dwell()** = true, **click()** is called repeatedly until **mouseUp()**.

click() calls the protected function **click_()**. If **click_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **click_()** is overridden in a user-derived class, the override function is executed.

dblClick

```
void dblClick();
```

The UI Framework calls **dblClick()** following a double click mouse event when the mouse is pointing to this object. **dblClick()** calls the protected function **dblClick_()**. If **dblClick_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **dblClick_()** is overridden in a user-derived class, the override function is executed.

dragStart

```
void dragStart(const ccIPair& startPos);
```

Called by the UI framework at the start of a dragging operation. *startPos* is the mouse starting position. **dragStart()** calls **dragStart_()** which can be overridden in a user-derived class to perform additional custom operations.

Parameters

startPos	Starting position of the mouse in screen coordinates.

dragStop

```
void dragStop(const ccIPair& startPos,
              const ccIPair& stopPos);
```

Called by the UI framework when a dragging operation stops. **dragStop()** calls **dragStop_()** which is overridden in **ccUIShapes::dragStop_()** where drag stop implementation is done.

Parameters

startPos	Starting position of the mouse in screen coordinates.
stopPos	Ending position of the mouse in screen coordinates.

dragAnimate

```
void dragAnimate(const ccIPair& startPos,
  const ccIPair& stopPos);
```

Called by the UI framework to display an outline of the dragged object(s) during the dragging operation. This allows the operator to more accurately choose the final position of the drag. **dragAnimate()** is called repeatedly during the drag so that the dragged object(s) appear to move in real time.

dragAnimate() calls **dragAnimate_()** which is overridden in **ccUIShapes::dragAnimate_()** where the dragging implementation is done.

Parameters

startPos	Starting mouse position of the drag in screen coordinates. For multiple calls to **dragAnimate()** during a dragging operation, *startPos* is always the same.
stopPos	The current mouse position of the drag in screen coordinates. Note that the dragging operation may not be finished, in which case this is an intermediate stopping position.

keyboard

```
void keyboard(const ccKeyboardEvent& ev);
```

Called by the UI framework in response to a keyboard event when the mouse is pointing to this object. For example, the user typing a keyboard key. **keyboard()** calls **keyboard_()** which can be overridden in a user-derived class to perform additional custom operations.

Parameters

ev	The keyboard event that initiated this call. See the **ccKeyboardEvent** reference page.

Protected Member Functions

show

```
virtual void show(bool parentChanged);
```

The UI Framework calls **show()** when the object's state changes to *visibleState (the object becomes visible)*. If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

If **show()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **show()** is overridden in a user-derived class, the override function is executed.

Parameters

parentChanged True if the state of the UI object's parent (or ancestor) has changed. False if the state of the object itself has changed.

hide

```
virtual void hide(bool parentChanged);
```

The UI Framework calls **hide()** when the object's state changes from *visibleState* (the object becomes not visible). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

If **hide()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **hide()** is overridden in a user-derived class, the override function is executed.

Parameters

parentChanged True if the state of the UI object's parent (or ancestor) has changed. False if the state of the object itself has changed.

enable

```
virtual void enable(States s, bool parentChanged);
```

The UI Framework calls **enable()** when the object's state changes to *enabledState* (the object becomes enabled). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

If **enable()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **enable()** is overridden in a user-derived class, the override function is executed.

Parameters

s The state change that caused the object to become enabled. Must be one of the **States** enums.

parentChanged True if the enable was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

disable

```
virtual void enable(States s, bool parentChanged);
```

The UI Framework calls **disable()** when the object's state changes from *enabledState* (the object becomes disabled). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

If **disable()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **disable()** is overridden in a user-derived class, the override function is executed.

Parameters

s	The state change that caused the object to become disabled. Must be one of the **States** enums.
parentChanged	True if the disable was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

select

```
virtual void select(States s, bool parentChanged);
```

The UI Framework calls **select()** when the object's state changes to *selectedState* (the object becomes selected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

If **select()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **select()** is overridden in a user-derived class, the override function is executed.

Parameters

s	The state change that caused the object to become selected. Must be one of the **States** enums.
parentChanged	True if the select was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

deselect

```
virtual void deselect(States s, bool parentChanged);
```

The UI Framework calls **deselect()** when the object's state changes from *selectedState* (the object becomes deselected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

If **deselect()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **deselect()** is overridden in a user-derived class, the override function is executed.

Parameters

s	The state change that caused the object to become deselected. Must be one of the **States** enums.
parentChanged	True if the deselect was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

mouseDown_ `virtual void mouseDown_();`

The UI Framework calls **mouseDown()** when the left mouse button is pressed down while pointing to this object (the object must be **clickable()**). **mouseDown()** then calls the protected function **mouseDown_()**. If **mouseDown_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseDown_()** is overridden in a user-derived class, the override function is executed.

mouseUp_ `virtual void mouseUp_();`

The UI Framework calls **mouseUp()** following a mouseDown when the left mouse button is released, or when the mouse pointer is moved off the object. **mouseUp()** then calls the protected function **mouseUp_()**. If **mouseUp_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseUp_()** is overridden in a user-derived class, the override function is executed.

mouseEnter_ `virtual void mouseEnter_();`

The UI Framework calls **mouseEnter()** when the mouse enters the UI object's screen space with a left mouse button up. **mouseEnter()** then calls **mouseEnter_()**. If **mouseEnter_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseEnter_()** is overridden in a user-derived class, the override function is executed.

idleMouseEnter_ `virtual void idleMouseEnter_();`

The UI Framework calls **idleMouseEnter()** when the mouse enters the UI object's screen space with a left mouse button up. **idleMouseEnter()** then calls **idleMouseEnter_()**. If **idleMouseEnter_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **idleMouseEnter_()** is overridden in a user-derived class, the override function is executed.

mouseLeave_ `virtual void mouseLeave_();`

The UI Framework calls **mouseLeave()** when the mouse is moved off the UI object or also if an enabled object was entered and it then becomes disabled. **mouseLeave()** then calls **mouseLeave_()**. If **mouseLeave_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseLeave_()** is overridden in a user-derived class, the override function is executed.

mouseMove_ `virtual void mouseMove_(const ccIPair& screenPos);`

The UI Framework calls **mouseMove()** repeatedly when the mouse is moving while pointing to this object. (For example, while resizing the object). Each call includes the current mouse position in screen coordinates. **mouseMove()** then calls the protected function **mouseMove_()**. If **mouseMove_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **mouseMove_()** is overridden in a user-derived class, the override function is executed.

Parameters

screenPos The current mouse position in screen coordinates.

click_ `virtual void click_();`

The UI Framework calls **click()** following a **mouseDown()** - **mouseUp()** sequence. Also, following a **mouseDown()**, if **dwell()** = true, **click()** is called repeatedly until **mouseUp()**.

click() calls the protected function **click_()**. If **click_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **click_()** is overridden in a user-derived class, the override function is executed.

dblClick_ `virtual void dblClick_();`

The UI Framework calls **dblClick()** following a double click mouse event when the mouse is pointing to this object. **dblClick()** calls the protected function **dblClick_()**. If **dblClick_()** is not overridden in a user-derived class, it does nothing and returns. No additional action is taken. If **dblClick_()** is overridden in a user-derived class, the override function is executed.

dragStart_ `virtual void dragStart_(const ccIPair& startPos);`

Called by **dragStart()**. Unless overridden in a user-derived class, this function does nothing. Provides a way for a user to perform custom operations at the start of a dragging operation.

See **dragStart()**.

Parameters

startPos Starting position of the drag in screen coordinates.

dragStop_

```
virtual void dragStop_(const ccIPair& startPos,
                       const ccIPair& stopPos);
```

Called by **dragStop()**. Unless overridden in a user-derived class, this function does nothing. Provides a way for a user to perform custom operations at the end of a dragging operation.

See **dragStop()**.

Parameters

startPos Starting position of the mouse in screen coordinates.

stopPos Ending position of the mouse in screen coordinates.

dragAnimate_

```
virtual void dragAnimate_(const ccIPair& startPos,
                          const ccIPair& curPos);
```

Called by **dragAnimate()**. Unless overridden in a user-derived class, this function does nothing. Provides a way for a user to perform custom operations during a dragging operation.

See **dragAnimate()**.

Parameters

startPos Starting mouse position of the drag in screen coordinates. For multiple calls to **dragAnimate()** during a dragging operation, *startPos* is always the same.

stopPos The current mouse position of the drag in screen coordinates. Note that the dragging operation may not be finished, in which case this is an intermediate stopping position.

mouseMiddle_

```
virtual void mouseMiddle_(const ccIPair& p,
                          MouseAction event, c_UInt32 keys);
```

Called by **mouseMiddle()**. Unless overridden in a user-derived class, this function does nothing. Provides a way for a user to perform custom operations during a dragging operation.

See **mouseMiddle()**.

Parameters

p Last position of the mouse in screen coordinates.

event Type of mouse event that caused this call. Must be one of the **ccMouseEvent::Event** enums (*eNone, eMovement, eDownClick, eDoubleClick, eUpClick, eDwell*).

keys Modifier keys depressed when the mouse action occurred. Must be one of the **ccMouseEvent::Key** enums (*eNoKey*, *eAlt*, *eControl*, *eShift*).

mouseRight_

```
virtual void mouseRight_(const ccIPair& p,
                         MouseAction event, c_UInt32 keys);
```

Called by **mouseRight()**. Unless overridden in a user-derived class, this function does nothing. Provides a way for a user to perform custom operations during a dragging operation.

See **mouseRight()**.

Parameters

p Last position of the mouse in screen coordinates.

event Type of mouse event that caused this call. Must be one of the **ccMouseEvent::Event** enums (*eNone, eMovement, eDownClick, eDoubleClick, eUpClick, eDwell*).

keys Modifier keys depressed when the mouse action occurred. Must be one of the **ccMouseEvent::Key** enums (*eNoKey*, *eAlt*, *eControl*, *eShift*).

keyboard_

```
virtual void keyboard_(const ccKeyBoardEvent& ev);
```

Called by **keyboard()**. Unless overridden in a user-derived class, this function does nothing. Provides a way for a user to perform custom operations in response to keyboard events.

See **keyboard()**.

Parameters

ev The keyboard event that initiated this call. See the **ccKeyboardEvent** reference page.

Notes

Keyboard events are sent directly to the UI object that is capturing mouse events.

front_

```
virtual void front_(ccUIObject* kid) = 0;
```

Called by **front()** when the object is moved to the front. (For example, when it is selected). This function can be overridden in a user-derived class to add custom functionality. If you do override, make sure your override function calls the parent's **front_()**.

Parameters

kid Pointer to the object moved to the front.

back_

```
virtual void back_(ccUIObject* kid) = 0;
```

Called by **back()** when the object is moved to the back. This function can be overridden in a user-derived class to add custom functionality. If you do override, make sure your override function calls the parent's **back_()**.

Parameters

kid Pointer to the object moved to the back.

Static Functions

uiMutex

```
static ccMutex& uiMutex();
```

Returns a reference to the global UI mutex.

Notes

If you need to lock this operation, lock the UI object's root mutex first.

orphanMutex

```
static ccMutex& orphanMutex();
```

Returns a reference to the global orphan mutex (that is, the mutex for objects without a parent).

isValid

```
static bool isValid(ccUIObject*, c_UInt32 key,
  const ccUIRootObject*);
```

Returns true if the UI object specified by the given address and key exists and has the specified root object as a parent.

Notes

A return value of true means that the first argument points to a valid object that has not been deleted or had an ancestor deleted, that has a valid parent, and all of whose ancestors each have a valid parent. The execution time for this function depends on the number of objects that exist at a given instant.

selectColor

```
static void selectColor(const ccColor& c);
static const ccColor& selectColor();
```

Specifies the *select* color for all UI shapes.

- `static void selectColor(const ccColor& c);`

  Sets the color to use when an object is selected. The color is used for all objects.

  Parameters

  c The color.

- `static const ccColor& selectColor();`

  Returns the color to use when an object is selected.

dragColor

```
static void dragColor(const ccColor& c);
static const ccColor& dragColor();
```

Specifies the *drag* color for all UI shapes.

- `static void dragColor(const ccColor& c);`

  Sets the color to use when an object is dragged. The color is used for all objects.

  Parameters

  c The color.

- `static const ccColor& dragColor();`

  Returns the color to use when an object is dragged.

Typedefs

MouseAction

```
typedef ccMouseEvent::Event MouseAction;
```

ccUIObject

ccUIPointIcon

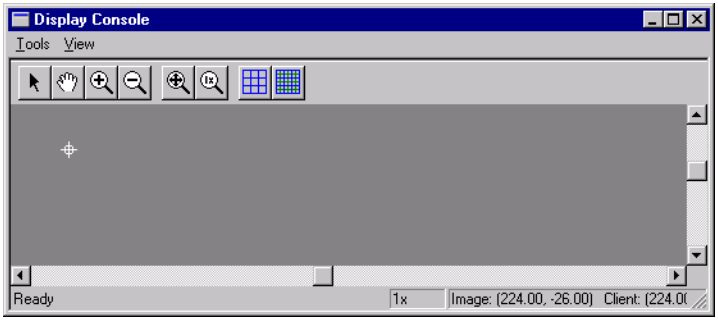
```
#include <ch_cvl/uishapes.h>

class ccUIPointIcon : public ccUIPointShapeBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a point icon you can display in a **ccDisplay** window where you can then position it using your mouse. The point icon size, shape, and orientation cannot be changed. The displayed point icon contains no handles for manipulation. The following is an example of a point icon that has been added to a **ccDisplayConsole** window.



ccUIPointIcon is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

You can left-click on the point icon, holding the button down to reposition it in a **ccDisplay** window. Point icons cannot be resized and have no manipulation handles.

After repositioning an point icon in a **ccDisplay** window, you will generally want to save the final point icon location for later use. Use the **ccUIShapes::pos()** getter function to obtain the final position of the point icon. If you later need to recreate the saved display, create a new default **ccUIPointIcon** and restore it to this saved location with **ccUIShapes::pos(cc2Vect)**.

Notes

ccUIPointIcon and any UI shape derived from **ccUIPointShapeBase** uses a coordinate space relative to its parent. When you add an object to a display window its parent is automatically set to one of the coordinate frame objects internal to the display.

Constructors/Destructors

ccUIPointIcon

```
ccUIPointIcon (ccUIObject* parent = NULL);
```

Creates a new manipulable point icon.

Parameters

parent	The point icon's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Protected Member Functions

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

isTouched `virtual bool isTouched (const cc2Vect& v, double scale);`

An override.

Returns true if the specified position (v) touches this point icon. Returns false otherwise.

Parameters

v	Position to be checked for proximity to the point icon. The position (*v*) is relative to its parent, in tablet coordinates. Multiply (*v*) by *scale* if you want the distance in pixels.
scale	Number by which to multiply (*v*) which is in tablet coordinates, to covert it to pixels.

ccUIPointIcon

ccUIPointSet

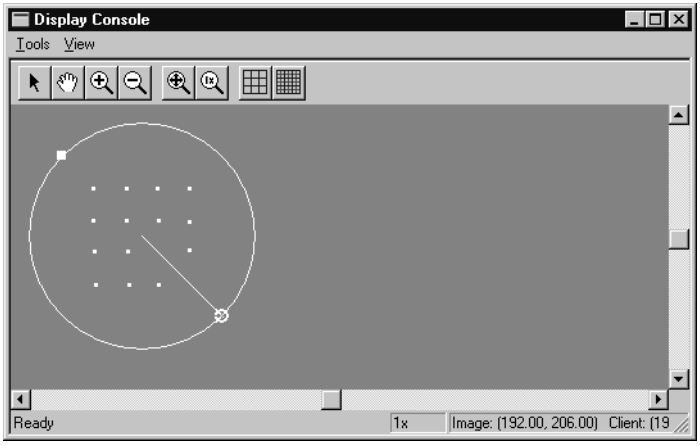
```
#include <ch_cvl/uishapes.h>

class ccUIPointSet : public ccUIManShape;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a point set (**ccPointSet**) that you can manipulate. When you add a **ccUIPointSet** object to a **ccDisplay** window the point set is displayed and can be moved about, resized, and rotated using your mouse. The following is an example of a point set that has been added to a **ccDisplayConsole** window.



The first point you specify in the point set you provide is used as the point set center. The second point you provide specifies the radius (p[0]-p[1]) of the enclosing circle shown in the example above. The display shows the enclosing circle and the radius line from p[0] to p[1]. The remainder of the points you specify make up the point set and are displayed as single pixel points.

ccUIPointSet is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIPointSet** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUIPointSet** has one resizing handle which allows you to expand or reduce the size of the enclosing circle. As you change the circle size, all points in the set change position proportionately.
 Rotation handle	Left-click on the rotation handle holding the button down and drag the handle to rotate the point set. The rotation handle is located at p[1].

After manipulating a point set in a **ccDisplay** window, you will generally want to retrieve the final point set state and use it in some way. You can use code similar to the following to retrieve the manipulated point set.

```
ccPointSet new_pointSet;
new_pointSet = ui_pointSet->pointSet();
```

Where **ui_pointSet** is the manipulated **ccUIPointSet** object.

Notes

ccUIPointSet and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIPointSet

```
ccUIPointSet(ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccPointSet**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The point set's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Enumerations

eLock

This enumeration lets you specify which handles are not visible when the object is selected. By using these locks, you can prevent a rectangle from being modified along different degrees of freedom.

Value	Meaning
eNoLocks	No locks. All manipulations allowed.
eSizeLock	The object cannot be resized.
eRotLock	The object cannot be rotated.

Public Member Functions

pointSet

```
void pointSet(const ccPointSet& pointSet);
const ccPointSet &pointSet() const;
```

- `void pointSet(const ccPointSet& pointSet);`

  Associated a **ccPointSet** with this manipulation object.

  Parameters

  pointSet The point set to make manipulable.

- `const ccPointSet& pointSet() const;`

  Gets the **ccPointSet** associated with this manipulation object.

updateHandles

```
virtual void updateHandles ();
```

Update all of my handles. Called when handles have been repositioned in the display.

showHandles

```
virtual void showHandles (bool show);
```

Make my handles visible/invisible depending on *show*, and the DOF locks. If the handles aren't created yet, create them.

Parameters

show True = show handles. False = do not show handles.

freeHandles

```
virtual void freeHandles ();
```

Deletes all handles.

locks

```
c_UInt32 locks() const;
void locks(c_UInt32 b);
```

- `c_UInt32 locks() const;`

  Returns the lock bits that indicate which resizing operations are not allowed for this object.

- `void locks(c_UInt32 b);`

  Sets the lock bits that indicate the resizing operations that are not allowed for this object.

  Parameters

  b The lock bits. May be any of the following ORed together:

  ccUIPointSet::eNoLocks
  ccUIPointSet::eSizeLock
  ccUIPointSet::eRotLock

  These lock values are described in *Enumerations* on page 3307.

setLocks

```
void setLocks(c_UInt32 b);
```

ORs the specified locks to the current locks.

Parameters

b The locks to OR with the current locks. May be any of the following ORed together:

ccUIPointSet::eNoLocks
ccUIPointSet::eSizeLock
ccUIPointSet::eRotLock

These lock values are described in *Enumerations* on page 3307.

clrLocks `void clrLocks(c_UInt32 b);`

Clears the locks specified in *b*. (ANDs the current locks with the complement of *b*)

Parameters

b The locks to clear. May be any of the following ORed together:

ccUIPointSet::eNoLocks
ccUIPointSet::eSizeLock
ccUIPointSet::eRotLock

These lock values are described in *Enumerations* on page 3307.

isTouched `virtual bool isTouched(const cc2Vect& pt, double scale);`

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt The position to test in the coordinate system used to draw the shape.

scale The scale to convert *pt* to pixels.

Protected Member Functions

pos_ `virtual ccPoint pos_ () const;`

`virtual void pos_ (const cc2Vect& pos);`

An override.

- `virtual ccPoint pos_ () const;`

  Returns the center position of the point set relative to its parent, in tablet coordinates.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the center position of the point set relative to its parent, in tablet coordinates.

  Parameters

  pos The new center position.

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal;* draws full point set *ccUIShapes::drawAbridged;* draws full point set *ccUIShapes::drawDrag;* draws full point set

Static Functions

touchDist

```
static float touchDist();

static void touchDist(float d);
```

- `static float touchDist();`

Gets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

- `static void touchDist (float d);`

Sets the distance from all instance of this object that is still considered a click on the object. The distance is specified in display coordinates.

Parameters

d	The distance from the object.

ccUIPointShapeBase

```
#include <ch_cvl/uishapes.h>

class ccUIPointShapeBase : public virtual ccUIShapes;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class is used as a base class for manipulable shapes that have icon-like behavior such as **ccUIRLEBuffer**, **ccUIIcon**, **ccUIPointIcon**, and **ccUILabel**.

Constructors/Destructors

ccUIPointShapeBase

```
ccUIPointShapeBase();
```

Creates a default manipulable object. You should never need to create an object of this class.

Public Member Functions

isTouched

```
virtual bool isTouched (const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

touchDist

```
float touchDist();
void touchDist(float d);
```

- `float touchDist();`

  Gets the distance from all instance of this object that is still considered a click on the object. The distance is specified in the coordinate system that the object is drawn in.

- `void touchDist(float d);`

  Sets the distance from all instance of this object that is still considered a click on the object. The distance is specified in the coordinate system that the object is drawn in.

  Parameters

  d The distance from the object.

Protected Member Functions

pos_

```
virtual ccPoint pos_ () const;
virtual void pos_ (const cc2Vect& pos);
```

An override.

- `virtual ccPoint pos_ () const;`

  Returns the center position of the UI shape relative to its parent, in tablet coordinates.

- `virtual void pos_ (const cc2Vect& pos);`

  Sets the center position of the UI shape relative to its parent, in tablet coordinates.

  Parameters

  pos The new center position.

Typedefs

ccUIInvisibleShape

```
typedef ccUIPointShapeBase ccUIInvisibleShape;
```

ccUIRectangle

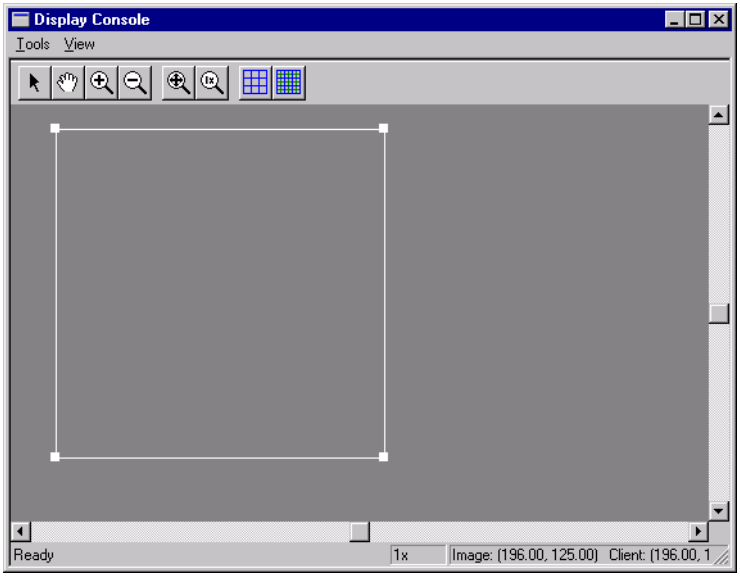
```
#include <ch_cvl/uishapes.h>

class ccUIRectangle : public ccUIGenRect;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates a rectangle (**ccRect**) that you can manipulate. When you add a **ccUIRectangle** object to a **ccDisplay** window a rectangle is displayed that can be moved about and resized using your mouse. The following is an example of a rectangle that has been added to a **ccDisplayConsole** window.



ccUIRectangle is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the **ccUIObject** and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

When you select interactive graphic objects in a **ccDisplay** window, *handles* appear as blue icons on the selected objects. You can grab these handles with the mouse to manipulate the objects. **ccUIRectangle** uses the following handles:

Handle	Description
 Resizing handle	Left-click on the resize handle holding the button down and drag the handle to resize the object. **ccUIRectangle** has four resizing handles which all operate in the same way.

After manipulating a rectangle in a **ccDisplay** window, you will generally want to retrieve the final rectangle state and use it in some way. You can use code similar to the following to retrieve the manipulated rectangle.

```
ccRect new_rect;
new_rect = ui_rect->rect();
```

Where **ui_rect** is the manipulated **ccUIRectangle** object.

Notes

ccUIRectangle and all UI shapes derived from **ccUIManShape** use a coordinate space relative to the tablet into which the shape is drawn.

Constructors/Destructors

ccUIRectangle

```
ccUIRectangle(ccUIObject* parent = NULL);
```

Creates a new manipulation object for a **ccRect**.

Notes

If you specify a parent, it must be derived from **ccUIPointShapeBase**. These include **ccUILabel**, **ccUIPointIcon**, **ccUIRLEBuffer**, and **ccUIIcon**.

Parameters

parent The rectangle's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

rect

```
void rect(const ccRect& r);
ccRect rect() const;
```

- `void rect(const ccRect& r);`

  Associates a **ccRect** with this manipulation object.

  Parameters

  r The rectangle to make manipulable.

  Throws

  ccShapesError::DegenerateShape
  r is degenerate.

  See **ccRect::degen()** for a description of degenerate rectangles.

- `ccRect rect() const;`

  Gets the **ccRect** associated with this manipulation object.

pelRect

```
void pelRect(const ccPelRect& r);
ccPelRect pelRect(bool enclosing = true) const;
```

- `void pelRect(const ccPelRect& r);`

  Sets the rectangle of this rectangle to the specified **ccPelRect**.

  Parameters

  r The pel rect to use to set the manipulable rectangle.

- `ccPelRect pelRect(bool enclosing = true) const;`

  Gets the pel rect that corresponds to this manipulable rectangle.

  Parameters

  enclosing If true, returns the largest **ccPelRect** that encloses this shapes corresponding **ccRect**.

  If false, returns the that largest **ccPelRect** smaller than the shape's **ccRect**.

Protected Member Functions

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

ccUIRLEBuffer

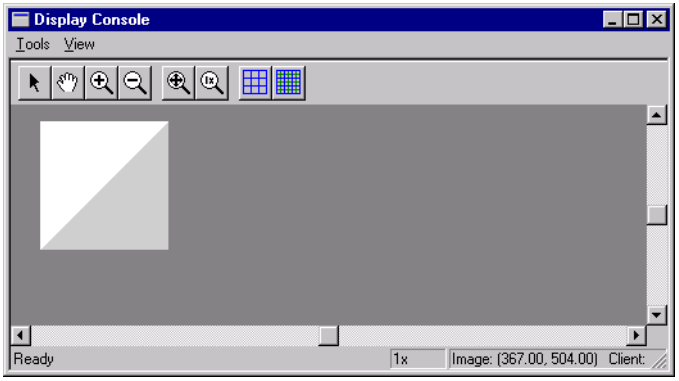
```
#include <ch_cvl/uishapes.h>

class ccUIRLEBuffer : public ccUIPointShapeBase;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	No

This class creates an icon from a **ccRLEBuffer** object. The icon size cannot be changed and it cannot be rotated. However, you can reposition the icon using your mouse. The displayed icon contains no handles for manipulation. The following is an example of a simple icon that has been added to a **ccDisplayConsole** window.



ccUIRLEBuffer is similar to **ccUIIcon** except **ccUIRLEBuffer** scales (when you zoom in on a **ccDisplay** window, the icon changes size) and **ccUIIcon** does scale. See the **ccUIIcon** reference page.

ccUIRLEBuffer is a member of a family of shapes that can be manipulated in a **ccDisplay** window. These classes are generally referred to as *UI shapes* or *interactive graphics* and are all derived from the **ccUIObject** and **ccUIShapes** base classes. Please see these base class reference pages for a discussion of how to manipulate UI shapes in a **ccDisplay** window using your mouse.

The **ccUIObject** and **ccUIShapes** base classes also provide member functions you can program to perform the same graphical manipulations as the mouse. When your program executes these functions, the shape displayed in a **ccDisplay** window changes in the same way it does when manipulated by the mouse. Please see the

ccUIObject and **ccUIShapes** reference pages for descriptions of these member functions. Also see the *Display Graphics* chapter of the *CVL User's Guide* for more information about using and displaying interactive graphics.

You can left-click on the icon holding the button down to reposition it in a **ccDisplay** window. Icons cannot be resized and have no manipulation handles.

After repositioning an icon you will generally want to retrieve the final icon state and use it in some way. You can use code similar to the following to retrieve the moved icon.

```
ccRLEBuffer new_rleBuf;
new_rleBuf = ui_rleBuf->rleBuffer();
```

Where **ui_rleBuf** is the moved **ccRLEBuffer** object.

Notes

ccUIRLEBuffer and any UI shape derived from **ccUIPointShapeBase** uses a coordinate space relative to its parent. When you add an object to a display window its parent is automatically set to one of the coordinate frame objects internal to the display.

Constructors/Destructors

ccUIRLEBuffer `ccUIRLEBuffer (ccUIObject* parent = NULL);`

Creates a new manipulation object for a **ccRLEBuffer**.

Parameters

parent	The run-length encoded buffer's parent frame. Typically, the display system will set up the parent for you when you add the shape to a display.

Public Member Functions

rleBuffer

```
void rleBuffer(const ccRLEBuffer& buf);

const ccRLEBuffer& rleBuffer() const;
```

- `void rleBuffer (const ccRLEBuffer& buf);`

Associates a **ccRLEBuffer** with this manipulation object.

Parameters

buf	The run-length encoded buffer to make manipulable.

Notes

Pixels whose value is **ccColor::passColor()** are not drawn.

- `const ccRLEBuffer& rleBuffer() const;`

  Gets the **ccRLEBuffer** associated with this manipulation object.

origin

```
void origin(const ccPoint& pt);
const ccPoint& origin() const;
```

- `void origin(const ccPoint& pt);`

  Specifies the origin within this image. The default is (0,0). The image is displayed at **absPos() – origin()**.

  Parameters

  pt The point

- `const ccPoint& origin() const;`

  Returns the origin with the point.

colorMap

```
void colorMap(cmStd vector<c_UInt8>& map);
const cmStd vector<c_UInt8>& colorMap() const;
```

- `void colorMap(cmStd vector<c_UInt8>& map);`

  Sets the color map to be associated with this run-length encoded image. The length of the map can range from zero to 256 elements.

  Parameters

  map The color map.

- `const cmStd vector<c_UInt8>& colorMap() const;`

  Gets the color map associated with this run-length encoded image.

useClientColor

```
void useClientColor(bool b);
bool useClientColor() const;
```

- `void useClientColor(bool b);`

  Specifies whether the run-length encoded image should use the color passed to the appropriate drawing routine for all instances of the client color set by **clientColor()** when the image is selected.

  Parameters

  b — If true, substitute the client color for all instances of the specified drawing color when the image is selected.

- `bool useClientColor() const;`

  Returns whether this image uses the client color when it is selected.

clientColor

```
void clientColor(c_UInt8 color);
c_UInt8 clientColor() const;
```

- `void clientColor(c_UInt8 color);`

  Specifies the pixel value (color) that should be replaced by the color passed to the appropriate drawing routine when the image is selected.

  Parameters

  color — The client color.

  Notes

  ccColor::isIndex() should be true for *color*.

- `c_UInt8 clientColor() const;`

  Returns the pixel value (color) that should be replaced by the color passed to the appropriate drawing routine when the image is selected.

passThrough

```
void passThrough(bool b);
bool passThrough() const;
```

- `void passThrough(bool b);`

  Sets whether the color specified by **passColor()** should not be drawn when drawing the run-length encoded buffer.

  Parameters

b	If true, the color specified by **passColor()** is not drawn.

- `bool passThrough() const;`

  Returns whether the color specified by **passColor()** should not be drawn when drawing the run-length encoded buffer.

passColor

```
void passColor(c_UInt8 color);
c_UInt8 passColor();
```

- `void passColor(c_UInt8 color);`

  Specifies the color in the image that should not be drawn when **passThrough()** returns true.

  Parameters

color	The passthrough color.

- `c_UInt8 passColor() const;`

  Returns the passthrough color.

dontScale

```
void dontScale(bool b);
bool dontScale() const;
```

- `void dontScale(bool b);`

  Specifies whether the run-length encoded image should be scaled to match the coordinate system in which it is drawn.

  Parameters

b	True if the image should not be scaled.

- `bool dontScale() const;`

  Returns whether the image should be scaled.

isTouched

```
virtual bool isTouched(const cc2Vect& pt, double scale);
```

Returns true if *pt* is within **touchDist()** of this object. This function is called automatically for you when you click the mouse on a shape. You should not need to call it yourself.

Parameters

pt	The position to test in the coordinate system used to draw the shape.
scale	The scale to convert *pt* to pixels.

Protected Member Functions

draw_

```
virtual void draw_ (
  ccUITablet& t,
  const ccColor& c,
  DrawMode m = drawNormal);
```

An override.

Called by **ccUIShapes::draw()** to draw the UI shape in a specified tablet.

Parameters

t	Tablet in which to draw the UI shape.
c	Color of the UI shape.
m	The drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

ccUIShapes

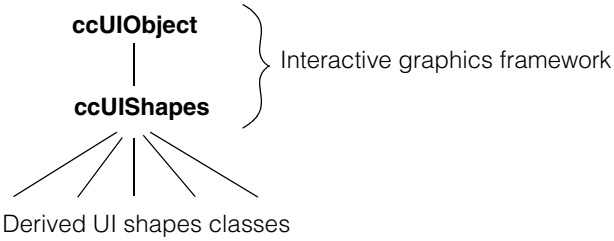
```
#include <ch_cvl/uifrmwrk.h>

class cmImport_cogdisp ccUIShapes : public ccUIObject;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccUIObject and **ccUIShapes** are base classes for the interactive graphics classes, also called UI shapes classes. These base classes make up the interactive graphics framework. See the following class derivation hierarchy.



Interactive graphics classes use the graphics classes defined in *shapes.h* and include fundamental shapes such as the line, rectangle, circle, ellipse, and others. These interactive classes provide an environment where shapes can be manipulated either graphically using a mouse and a **ccDisplay** window, or with your program using member functions from the framework base classes. Please see the **ccUIObject** reference page for additional functions that comprise the interactive graphics framework.

Member Function Summary

Use the following summaries as a guide to the **ccUIShapes** classes, and as a learning tool. The class member functions can be divided into the following functional groups:

Framework-only (these are overrides)	Relationships	Information
shapes() **tablet()**	**parent()** **closerSib()** **fartherSib()** **frontKid()** **backKid()** **numKids()**	**whoIsTouched()**^{v} (an override) **getGraphicProps()**^{v} **isTouched()**^{v} **pos()** **absPos()**

(**v**) = Virtual function

You should not call *Framework-only* classes in your program. *Relationships* and *Information* refer to the object from which they are called.

Attributes	Commands	Protected virtual functions	Protected virtual function overrides
props()	**draw()** **move()** **update()**	**draw_()** **move_()** **pos_()** **redim_()**	**show()** **hide()** **select()** **deselect()** **mouseEnter_()** **idleMouseEnter_()** **mouseLeave_()** **dragStop_()** **dragAnimate_()** **front_()** **back_()**

Attributes allow you to tailor objects for your application. *Commands* are functions you can call when you are manipulating graphics under program control. *Virtual functions* are place holders for overrides in derived classes. *Virtual function overrides* are functions implemented in this class that override place holders in **ccUIObject**. These functions can be overridden again in other derived classes to add additional functionality. However, these downstream overrides should include a call to the base class function being overridden since required functionality is implemented here.

For additional information about interactive graphics classes, see the *Displaying Graphics* chapter of the *CVL User's Guide*.

Constructors/Destructors

This is a pure virtual class. Only classes derived from this class use the constructors and destructors.

~ccUIShapes

```
virtual ~ccUIShapes();
```

Destructor. Destroys the UI shape and removes it from its parent's list of kids. Scans the UI shape's list of kids, deletes those whose autoDelete flag is set, and sets the parent of the remaining kids to NULL.

Enumerations

DrawMode

This enumeration specifies the drawing mode for shapes. It is used to specify the appearance of a shape during various drawing operations.

Value	Meaning
drawNormal	Normal shape appearance
drawAbridged	Abridged appearance for faster drawing
drawDrag	Appearance during a drag operation

Public Member Functions

shapes

```
virtual ccUIShapes* shapes(bool canFail);
```

RTTI framework operation providing base-to-derived type conversion for framework classes. Returns the *this* pointer if the object is an instance of **ccUIShapes**, otherwise 0. Relatively lightweight as compared to *dynamic_cast<>*.

Parameters

canFail — Specifies whether or not the operation can fail. False asserts that the conversion can be done and causes a valid pointer always to be returned. A non-zero value causes NULL to be returned if the conversion cannot be done.

closerSib

```
virtual ccUIObject* closerSib();
```

Returns the sibling object added to the parent object just after this object. If there is none, it returns 0.

This is an override.

fartherSib

```
virtual ccUIObject* fartherSib();
```

Returns the sibling object added to the parent object just before this object. If there is none, it returns 0.

This is an override.

frontKid

```
virtual ccUIObject* frontKid();
```

Returns the child object of this shape that was last added to the parent object. If there is none, it returns 0.

This is an override.

backKid

```
virtual ccUIObject* backKid();
```

Returns the child object of this shape that was first added to the parent object. If there is none, it returns 0.

This is an override.

See **ccUIObject::closerSib** on page 3272 for an example of code that uses this function.

numKids

```
virtual c_Int32 numKids();
```

Gets the number of kids of the UI object.

This is an override.

tablet

```
virtual ccUITablet& tablet();
```

Gets the tablet in which the UI shape is to be drawn. Shapes do not have their own tablet, rather they are drawn in their parent's tablet. If the UI shape's parent is also a shape, then the UI shape is drawn in the grandparent's tablet, and so on until an ancestor is found that is a frame.

This is an override.

whoIsTouched

```
virtual ccUIObject* whoIsTouched(const ccIPair& p);
```

Returns a pointer to the front-most, visible UI object touching the specified point that is either the UI shape or one of its kids. Returns NULL if the point touches neither the shape nor any of its kids. The point is translated from the parent's scope to the UI shape's tablet coordinates. Calls the private non-virtual **whoIsTouched()** function to do the work.

Parameters

p Position to be checked for proximity to UI objects.

Notes

The point is specified in the scope coordinates of the UI object's parent. For the root object, these are screen coordinates. For shapes, they are the same as the UI object's tablet coordinates.

Notes

Only framework classes should override this method. Override **ccUIFrame::isTouched()** and **ccUIShapes::isTouched()** in classes you derive to specify how particular objects are touched.

---

props

```
void props(const ccGraphicProps &props);

const ccGraphicProps &props() const;
```

- `void props(const ccGraphicProps &props);`

  Sets the graphic properties of the UI shape. Triggers updating, erasing, and redrawing of the UI shape.

  Parameters

  props The new graphics properties.

  Note that **ccGraphicProps::penColor()** specifies the object's deselected color.

- `const ccGraphicProps &props() const;`

  Gets the graphic properties of the UI shape. To get the graphic properties of the current rendering state, depending on whether or not the shape is selected, use **getGraphicProps** instead.

---

color

```
void color(const ccColor& c);

const ccColor& color() const;
```

Note This function is deprecated and is retained for backward compatibility only. All new code should use **props()**.

- `void color(const ccColor& c);`

  Sets the color of the UI shape. You can use this function to override **getColor** to return a different color. This function also allows you to set a shape's color without deriving a new class. This function applies the new color to the UI shape's graphical properties, triggering an update but no erasing or redrawing of the shape.

  Parameters

c	The color.

- `const ccColor& color() const;`

  Gets the color of the UI shape. Used by the **getColor** function.

isTouched

`virtual bool isTouched(const c2Vect& v, double scale);`

Returns false. Derived classes override this function to return true if the specified position (v) touches the UI shape. For example, the touch criteria for a circle may be whether the point falls either inside the circle or on its border, whereas for a line the touch criteria would depend on the positions of its endpoints.

Parameters

v	Position to be checked for proximity to the UI shape. The position (*v*) is relative to the UI shape's parent, in tablet coordinates. Multiply (*v*) by *scale* if you want the distance in pixels.
scale	Number by which to multiply (*v*) which is in tablet coordinates, to covert it to pixels.

getGraphicProps

`virtual ccGraphicProps getGraphicProps() const;`

Gets the current graphic properties of the UI shape. If the shape is not selected, **ccGraphicProps::penColor()** returns its default color or the color you have set. This is the deselected color.

If the shape is selected, **ccGraphicProps::penColor()** returns **ccUIObject::selectColor()**.

Notes

To get the current **ccGraphicProps** object associated with the shape, derived classes overriding the **draw_** function should call this function in their implementations rather than calling **props**. The base implementation sets the color of the shape based on its selected state. If overloading this function, you may want to call this base function first to maintain the selection color behavior.

getColor

```
virtual ccColor getColor() const;
```

Note This function is deprecated and is retained for backward compatibility only. All new code should use **props()**.

If UI the shape is selected, this function returns the color associated with selected shapes (as returned by **selectColor()**), otherwise that associated with deselected shapes (as returned by **color()**). Derived classes can override the function to return the color used for normal drawing.

parent

```
virtual ccUIObject* parent();

void parent(ccUIObject *newParent);
```

Retrieves or sets the parent of the UI shape.

- `virtual ccUIObject* parent();`

  Retrieves the parent (runtime owner) of the UI shape. Implements the pure virtual function of the base class.

- `void parent(ccUIObject *newParent);`

  Sets or changes the parent of the UI shape.

  Parameters

  newParent The new parent. Must be either a NULL pointer or a pointer to a an instance of **ccUIShapes** or **ccUIShapesFrame**.

  Throws

  ccUIError::BadParams

  newParent is pointing to this UI shape. The parent must be a different UI object.

  Notes

  When assigning a parent to interactive shapes using **ccUIShapes::parent(newParent)**, all children must be on the same drawing layer as the parent. Cognex recommends setting the parent of an interactive shape before adding the shape to the display. Changing a shape's parent once the shape has been added to the display will result in lower performance when the graphics are re-rendered.

  For optimal rendering performance, do not change the drawing layer or parent of an interactive shape after adding the shape to the display console.

pos

```
ccPoint pos() const;
void pos(const cc2Vect& v);
```

Returns the center location of the UI shape relative to its parent in tablet coordinates.

pos() calls **pos_()**, a protected virtual function you can override in your own derived class to customize this method.

- `ccPoint pos() const;`

  Returns the center location of the UI shape.

- `void pos(const cc2Vect& v);`

  Sets the center location of the UI shape relative to its parent in tablet coordinates.

  Parameters

  v The position.

absPos

```
ccPoint absPos();
```

Returns the center location of the UI shape in tablet coordinates. This is the same as **pos()** if the object is not grouped.

move

```
void move(const cc2Vect& del);
```

Moves the UI shape by adding **del.x()** and **del.y()** to the current position.

move() calls **move_()**, a protected virtual function you can override in your own derived class to customize this method.

Parameters

del The distance to move.

update

```
void update(
  bool kidsToo = true,
  bool eraseFirst = true,
  const ccUIRect* activeScopes = NULL,
  ccUITablet::Layers l = ccUITablet::eUnspecifiedLayer);
```

Redraws the UI shape.

Parameters

kidsToo If true, update all descendants (default = true).

eraseFirst	If true, erase the UI shape before redrawing. Use false if you know that only the color has changed (default = true).
activeScopes	Set of rectangles defining the active scopes of the UI shape.
l	Tablet layer in which to redraw the UI shape.

draw

```
void draw(ccUITablet& t, const ccColor& c,
  DrawMode m = drawNormal);
```

Draws the UI shape in the specified tablet, using the specified graphic properties, and in the specified drawing style.

draw() calls **draw_()**, a protected virtual function you can override in your own derived class to customize this method.

Parameters

t	Tablet in which to draw the UI shape.
c	Graphic properties used for drawing.
m	Drawing mode. Must be one of: *ccUIShapes::drawNormal* *ccUIShapes::drawAbridged* *ccUIShapes::drawDrag*

Notes

Do not use this function to cause a UI shape to appear on the screen or to reflect changes in state. Use **condVisible** and **update** for those purposes. This function is provided as a public member so that clients can render the UI shape in a tablet other than the shape's own tablet (for example, to make a sketch).

Protected Member Functions

This section describes only the protected member functions of **ccUIShapes** that are also virtual.

front_

```
virtual void front_(ccUIObject* kid);
```

An override.

Called by **ccUIObject::front()** when the object is moved to the front. (For example, when it is selected). If you override this function, make sure your override function calls the parent's **front_()**.

Parameters

kid	Pointer to the object moved to the front.

back_ `virtual void back_(ccUIObject* kid);`

An override.

Called by **ccUIObject::back()** when the object is moved to the back. If you override this function, make sure your override function calls the parent's **back_()**.

Parameters

kid Pointer to the object moved to the back.

show `virtual void show(bool parentChanged);`

The UI Framework calls **ccUIObject::show()** when the object's state changes to *visibleState (the object becomes visible).* If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

This is an override function that displays the changed object. If **show()** is overridden in a user-derived class, the override function should call **show()** in the parent class.

Parameters

parentChanged True if the state of the UI object's parent (or ancestor) has changed. False if the state of the object itself has changed.

hide `virtual void hide(bool parentChanged);`

The UI Framework calls **ccUIObject::hide()** when the object's state changes from *visibleState* (the object becomes not visible). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

This is an override function that displays the changed object. If **hide()** is overridden in a user-derived class, the override function should call **hide()** in the parent class.

Parameters

parentChanged True if the state of the UI object's parent (or ancestor) has changed. False if the state of the object itself has changed.

select

```
virtual void select(States s, bool parentChanged);
```

The UI Framework calls **ccUIObject::select()** when the object's state changes to *selectedState* (the object becomes selected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

This is an override function that implements the *select* procedure and calls **update** in the case of display color or other appearance changes. If **select()** is overridden in a user-derived class, the override function should call **select()** in the parent class.

Parameters

s	The state change that caused the object to become selected. Must be one of the **ccUIObject::States** enums.
parentChanged	True if the select was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

deselect

```
virtual void deselect(States s, bool parentChanged);
```

The UI Framework calls **ccUIObject::deselect()** when the object's state changes from *selectedState* (the object becomes deselected). If the change was caused by the parent object, *parentChanged* = true. If the change was caused by the object itself, *parentChanged* = false.

This is an override function that implements the *deselect* procedure and calls **update** in the case of display color or other appearance changes. If **deselect()** is overridden in a user-derived class, the override function should call **deselect()** in the parent class.

Parameters

s	The state change that caused the object to become selected. Must be one of the **ccUIObject::States** enums.
parentChanged	True if the select was caused by a parent (or ancestor) state change. False if the state of the object itself has changed.

dragAnimate_

```
virtual void dragAnimate_(const ccIPair& p1,
  const ccIPair& p2);
```

An override that implements the drag display for all shapes.

Draws the drag animator, the graphical representation of a UI shape while it is being dragged, at the current position. Implements the protected virtual function of the base class to handle generic dragging for all shapes. The drag animator is the *drawDrag* version of the shape itself, drawn in the *dragColor*.

Parameters

p1	Starting position in screen coordinates.

p2 Current position of the drag in screen coordinates.

Notes

If the current position is the Cancel position, the drag animator is drawn at the starting position. Do not use this function to implement non-generic dragging. This function requires derived classes to fully support the **pos()** and **pos(ccVect)** functions.

dragStop_

```
virtual void dragStop_(const ccIPair& p1,
  const ccIPair& p2);
```

An override.

Updates the state of the UI shape based on its final drag position. Implements the protected virtual function of the base class to handle generic dragging for all shapes. The drag animator is the *drawDrag* version of the shape itself, drawn in the *dragColor*.

Parameters

p1 Starting position in screen coordinates.

p2 Ending position of the drag in screen coordinates.

Notes

It is good practice to define certain drag positions to mean Cancel, so that **dragStop_** can do nothing in those cases. Do not use this function to implement non-generic dragging. This function requires derived classes to fully support the **pos()** and **pos(ccVect)** functions.

mouseEnter_

```
virtual void mouseEnter_();
```

An override. See **ccUIObject::mouseEnter()**.

Implements mouse enter when the mouse enters the UI object's screen space with a left mouse button held down. Switches the mouse cursor between the shape used for the UI object and a default cursor shape.

mouseLeave_

```
virtual void mouseLeave_();
```

An override. See **ccUIObject::mouseLeave()**.

Implements mouse leave when the mouse is moved off the UI object, or also, if an enabled object was entered and it then becomes disabled. Switches the mouse cursor between the shape used for the UI object and a default cursor shape.

idleMouseEnter_

```
virtual void idleMouseEnter_();
```

An override. See **ccUIObject::idleMouseEnter()**.

Implements idle mouse enter when the mouse enters the UI object's screen space with the left mouse button up. Switches the mouse cursor between the shape used for the UI object and a default cursor shape.

pos_

```
virtual ccPoint pos_() const = 0;

virtual void pos_(const cc2Vect& v) = 0;
```

Called by **pos()** to get/set the object's center location. These virtual functions are implemented in a derived class.

- `virtual ccPoint pos_() const = 0;`

  Gets the center location of the UI shape relative to its parent in tablet coordinates. Derived classes implement this pure virtual function. Call the like-named public member function in your own programs.

- `virtual void pos_(const cc2Vect& v) = 0;`

  Sets the center location of the UI shape relative to its parent in tablet coordinates. Derived classes implement this pure virtual function. Call the like-named public member function in your own programs.

  Parameters

v	Center location of the UI shape relative to its parent in tablet coordinates.

  Notes

  Derived classes should override this function only if standard dragging support is required.

draw_

```
virtual void draw_(ccUITablet& t, const ccColor& c,
  DrawMode m = drawNormal);
```

Called by **draw()** to draw the UI shape in a specified tablet. This function should be overridden and implemented in derived classes. You should call **draw()** in your programs.

Parameters

t	Tablet in which to draw the UI shape.

c Color of the UI shape.

m The drawing mode. Must be one of:
ccUIShapes::drawNormal
ccUIShapes::drawAbridged
ccUIShapes::drawDrag

Notes

The color is based on the value returned by **getColor**. Derived classes that override this function and want to use the graphic properties should call **getGraphicProps** in their implementations.

move_ `virtual void move_(const cc2Vect& v);`

Called by **move()** to move the UI shape. This function should be overridden and implemented in derived classes. You should call **move()** in your programs.

Parameters

v Ending position of the move.

Notes

This function is called just before the UI shape moves to a new position. The **pos** function supplies the current position, the argument the ending position. This function is called if the position is changed via the **pos(const cc2Vect&)** overload and the ending position is different from the current position.

redim_ `virtual void redim_();`

This virtual function has no effect. Derived classes override the function to take some action when any of the UI shape's dimensions (for example, size, orientation, or position) changes.

Notes

This function is called just after the shape's dimensions change via a call to **pos**, **move**, or any shape-specific geometry setter.

ccUISketch

```
#include <ch_cvl/uisketch.h>

class ccUISketch;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A sketch is a list of zero or more drawing commands used to draw shapes into a tablet. You typically draw shapes into a tablet using the **ccUITablet::draw()** functions. When you draw shapes into a tablet, the shapes are actually accumulated in the tablet's sketch. When you create a tablet, a sketch is automatically created, so in most cases you do not need to create a sketch object yourself.

After you have drawn the shapes into a tablet, you use **ccUITablet::sketch()** to obtain the sketch, and then call **ccDisplay::drawSketch()** to draw the shapes accumulated in the sketch onto the display.

You can use sketch marks (see **ccUISketchMark** on page 3341) to delimit subsketches, or components of sketches.

Note It is not possible to archive a **ccUISketch** within another **ccUISketch**.

Constructors/Destructors

ccUISketch

```
ccUISketch();

ccUISketch(const ccUISketch&);

~ccUISketch();
```

- `ccUISketch();`

  Creates an empty sketch.

- `ccUISketch(const ccUISketch& s);`

  Copy constructor.

  Parameters

s	The **ccUISketch** to copy.

- `~ccUISketch();`

  Destructor.

Operators

operator+= `ccUISketch& operator+=(const ccUISketch& s);`

Adds the sketch *s* to the end of this sketch.

Parameters

s	The sketch to add to the end of this sketch.

operator= `ccUISketch& operator=(const ccUISketch& s);`

Assignment operator. Assignment the value of *sketch* to this object.

Parameters

s	The sketch to assign to this one.

operator+ `ccUISketch operator+(const ccUISketch& s) const;`

Returns the concatenation of this object and the supplied sketch.

Parameters

s	The sketch to add to this one.

Public Member Functions

isNull `bool isNull() const;`

Returns true if this sketch is empty, false otherwise.

mark `ccUISketchMark mark() const;`

Returns the sketch mark at the end of this sketch.

subSketch

```
ccUISketch subSketch(ccUISketchMark start,
  ccUISketchMark end) const;
```

Returns a new sketch from *start* up to (but not including) *end*.

Parameters

start	The beginning of the subsketch.
end	The end of the subsketch.

Notes

If *start* is greater than or equal to *end*, this method returns an empty sketch.

remove

```
ccUISketch& remove(ccUISketchMark start,
  ccUISketchMark end);
```

Removes a portion of the sketch from *start* up to (but not including) *end*. Returns this sketch minus the specified portion.

Parameters

start	The beginning of the portion to remove.
end	The end of the portion to remove.

Notes

If *start* is greater than or equal to *end*, this sketch is returned unchanged.

insert

```
ccUISketch& insert(const ccUISketch& s,
  ccUISketchMark beforeHere);
```

Inserts the sketch *s* into this sketch before the specified position.

Parameters

s	The sketch to insert.
beforeHere	The place before which to insert the sketch.

erase

```
ccUISketch& erase();
```

Erases this sketch and returns an empty sketch.

Notes

This method also resets the graphical properties back to the default.

eraseAndDelete

```
ccUISketch& eraseAndDelete();
```

Erases this sketch, releases any accumulated storage, and returns an empty sketch.

Notes

This method also resets the graphical properties back to the default.

ccUISketchMark

```
#include <ch_cvl/uisketch.h>

class ccUISketchMark;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

You can use sketch marks to identify parts of sketches. A sketch mark is a value that identifies the beginning or end of a component in a sketch.

See **subSketch()**, **insert()**, and **remove()** in **ccUISketch** on page 3337 for information on methods you can use to access and manipulate components of sketches.

Constructors/Destructors

ccUISketchMark `ccUISketchMark();`

Creates a new mark.

Operators

operator== `bool operator==(ccUISketchMark &m) const;`

Returns true if this mark and the mark *m* are the same mark.

Parameters

m	The other mark.

operator!= `bool operator!= (ccUISketchMark &m) const;`

Returns true if this mark and the mark *m* are not the same mark.

Parameters

m	The other mark.

ccUISketchMark

ccUITablet

```
#include <ch_cvl/uitablet.h>

class cmImport_cogdisp ccUITablet;
```

Class Properties

Copyable	Yes
Derivable	Not intended
Archiveable	No

The **ccUITablet** class can be used in two distinct ways:

1. It is used as a base class for the **ccDisplay** class hierarchy:
   - To provide the interface for rendering non manipulable graphics (for example, static graphics) into the display.
   - Provides helper functions to manage the display content and to refresh the display of interactive graphics.
2. It can be instantiated directly and used for recording, a procedure where graphics operations are recorded into a tablet. No display-derived class is necessary for recording, meaning you can construct a **ccUITablet** for just recording with no display. Recordings can later be passed to a **ccDisplay** via **ccUISketch** for playback where the recordings are rendered into the display.

Because **ccUITablet** has this dual role, some of its member functions apply to only one particular use. Where this is the case, member functions contain comments to help clarify the intended use. For more about using this class see the discussion on page 3356.

Notes

ccUITablet is not multi-thread safe. If your application is multi-threaded you are responsible for controlling proper access to this class.

Scopes are a **ccUITablet** internal property used by Cognex developers when creating **ccUITablet**-derived display classes. You may see scopes mentioned in this reference material, but you do not have to be concerned about scopes.

Coordinate Spaces

When you record graphics into a tablet the graphics are specified in tablet space, a left-handed coordinate system. Later when you display a tablet you specify the display coordinate space; *eDisplayCoords*, *eImageCoords*, or *eClientCoords*. If you display in client coordinates, tablet (0,0) equals client (0,0). If you display in image coordinates, tablet (0,0) equals image (0,0), and so on.

See the **ccDisplay** reference page for more information regarding coordinate spaces for display windows.

Z-Order

When recording graphics and other components into a tablet's sketch list, new graphics are appended to the list end. Playback starts at the list beginning. Therefore, operations added first are rendered in the display first.

Using ccUITablet

ccUITablet describes a drawing environment for drawing shapes such as the ones described in the file *shapes.h*. In typical use, you draw shapes using **ccUITablet::draw()** or one of the specific shape-drawing functions, such as **ccUITablet::drawCircle()**. These shapes are accumulated into a sketch (see **ccUISketch** on page 3337) which is then drawn into a display derived from **ccDisplay** such as **ccDisplayConsole** using **ccDisplay::drawSketch()**.

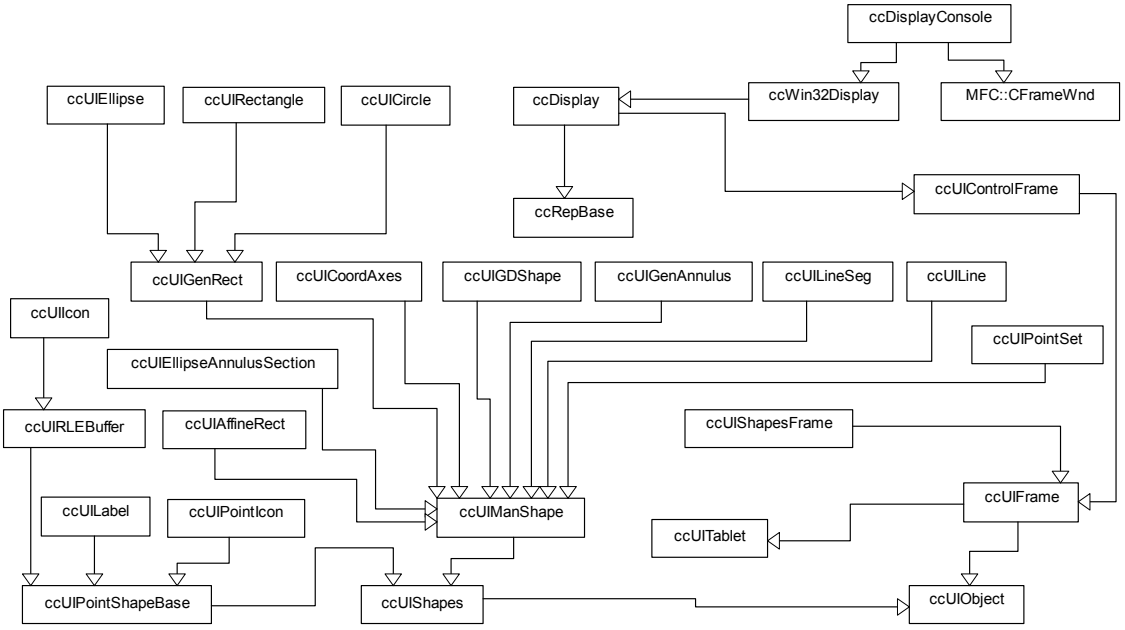
You can draw either into the overlay layer or the image layer. Whenever you draw into the image layer the display must be rerendered, adding time to the application. Drawing into the overlay layer, on the other hand, does not require the image layer to be rerendered, enabling your application to run faster.

By default, graphics are placed in the image layer. On some display systems the overlay layer must be explicitly enabled, and may not be enabled by default. Graphics added to an overlay layer that has not been enabled are not displayed

Display type	Overlay default	Description
ccWin32Display	Disabled	Call **ccWin32Display::enableOverlay()** to enable.
ccDisplayConsole	Disabled	

All of the shape classes in *shapes.h* and many of the drawing functions in **ccUITablet** require you to specify locations. These locations do not refer to any particular coordinate system. Rather, you specify the coordinate system to use with a tablet's sketch when you call **drawSketch()**. To draw different shapes in different coordinate systems, use different drawing tablets.

The following figure shows the interactive graphics class derivation hierarchy:



Constructors/Destructors

ccUITablet

```
ccUITablet();

virtual ~ccUITablet();
```

For recording graphical operations use the default constructor. This is the most common use. A **ccUITablet** is used for display when you construct a derived display class from **ccDisplay**. For example, a **ccWin32Display** or **ccDisplayConsole**. Scopes are not required to record into a tablet.

Direct derivation from this class is not supported. Only derivation through the **ccDisplay** class hierarchy is supported.

- `ccUITablet();`

  Default constructor. Creates a tablet for recording drawing commands. The scope is NULL, recording is enabled, drawing is disabled, the tablet does not have a parent, and an empty sketch is automatically created.

Notes

Except for the initial drawing and recording states, this is equivalent to **ccUITablet(0,0)**. You can use a default-constructed tablet for drawing after setting the scope and enabling drawing.

Use this constructor to create a tablet for recording graphics into a sketch. This sketch can then be passed to a display for rendering.

~ccUITablet

```
virtual ~ccUITablet();
```

Destructor. Erases the tablet from the screen and removes the tablet from the parent's list of kids. Sets all of its kids' parent to NULL.

Enumerations

Layers

```
enum Layers;
```

An enumeration of supported layers used for recording into a sketch or drawing into a display.

Notes

The **overlaySupported()** static function can be used to determine if your display supports the overlay layer.

Some platforms require you to enable the overlay layer for graphics to appear. See the derived class reference documentation for information about enabling overlay layers.

Value	Meaning
eImageLayer = 0	Draw shapes into the image layer.
eOverlayLayer = 1	Draw shapes into the overlay layer. For Cognex frame grabbers which use Windows 2000 as the host OS, you must explicitly enable the overlay layer via **ccWin32Display::enableOverlay(ccWin32Display::eOverlayPlane)**.
eNumLayers = 2	Number of layers used.
eUnspecifiedLayer = *eNumLayers*	The layer is not specified. Typically, this pseudo-layer is used in recording and playback. It is also used for sub-sketches. When you call **draw()** to draw a subsketch, you specify the layer into which otherwise unspecified shapes are drawn.

FillType

```
enum FillType;
```

This enumeration lets you specify how **fill()** fills an area with color.

Value	Meaning
eFillBorder	Fills the area until it reaches a specified stop color.
eFillSurface	Fills the area until it reaches a color other than the stop color.

InterpolationModes

```
enum InterpolationModes
```

This enumeration defines the type of display interpolation. It affects how the image is rendered when magnification is applied.

Notes

This functionality is only supported for **ccPelBuffer<c_UInt8>** images.

Value	Meaning
eNone	No display interpolation is performed. This option provides the fastest display, but the display can appear blocky or jagged when used to display zoomed images.
eBilinear	Uses bilinear interpolation to compute the value of each display pixel. This option produces a blurry-appearing display but requires only a modest additional amount of time for display. If you specify *eBilinear*, the displayed image is reduced in size by 2 pixels on each side.
eHighPrecision	Uses a high-precision interpolation method that considers additional pixel values. This option produces a smoothed image that does not appear blurry, but requires the most additional time for display. If you specify *eHighPrecision*, the displayed image is reduced in size by 3 pixels on each side.

Public Member Functions

interpolation

```
void interpolation(InterpolationModes interpolationMode);
InterpolationModes interpolation()const;
```

Interpolation mode is intended for use by derived classes (for example, **ccDisplay**) when a **ccUITablet** is being used as a base class for display.

Interpolation mode is not intended to be used when recording into a **ccUITablet**.

- `void interpolation(InterpolationModes interpolationMode);`

Sets the display interpolation mode for this display object. If you specify a value of *ccUITablet::eNone*, no interpolation is performed. If you specify one of the other interpolation modes and the image is displayed at a scale greater than 1.0, the specified interpolation mode is used to compute the value for each display pixel.

In all cases, nearest-neighbor sampling is used to compute the value of each display pixel when the scale is less than or equal to 1.0.

When you change the interpolation mode, the display is updated.

Parameters

interpolationMode

The interpolation method to use. Must be one of the following values:

ccUITablet::eNone
ccUITablet::eBilinear
ccUITablet::eHighPrecision

Notes

Display interpolation is not performed on images smaller than 8x8 pixels.

- `InterpolationModes interpolation() const;`

Returns the display interpolation mode for this tablet. This function returns one of the following values:

ccUITablet::eNone (default)
ccUITablet::eBilinear
ccUITablet::eHighPrecision

sketch

```
const ccUISketch& sketch() const;

ccUISketch& sketch();
```

Returns a display list of static graphics and other graphical elements (for example, filling areas) created by recording. Typically, you pass the sketch to **ccDisplay::drawSketch()** to draw the contents of a sketch.

When using a default constructed **ccUITablet**, recorded components are placed into this sketch. When this **ccUITablet** is being used as part of a display system and recording is on, graphical components are recorded into this sketch.

- `const ccUISketch& sketch() const;`

  Returns a read-only sketch.

- `ccUISketch& sketch();`

  Returns a read-write sketch. This version allows you to assign a sketch.

encloseRect

```
ccUIRect encloseRect(
  const ccUISketch& sketch,
  const ccPoint& point,
  Layers layer = eImageLayer) const;
```

Returns the enclosing rectangle of the sketch if drawn in the specified layer at the specified point, of this tablet. The returned rectangle is in scope coordinates, without regard to clipping by the current scope boundaries.

Parameters

sketch	The sketch enclosed by the returned rectangle.
point	The point in this tablet where the sketch is located.
layer	The layer in this tablet where the sketch is located.

Notes

Any components of *sketch* that explicitly use a layer other than that specified in *layer* are ignored. Components of a sketch generally use *eUnspecifiedLayer*.

The enclosing rectangle of all changes made to a tablet by drawing functions can be recorded. The enclosing rectangle is computed assuming that the scope is infinite, for example, without regard to the current actual boundaries of the scope's window. Some drawing functions, notably **set (ccColor)** and **setComplement()**, have an infinite enclosing rectangle. One exception is **draw (ccLine**, ...**)** which theoretically has an infinite enclosing rectangle, but is drawn as a line segment clipped to the active area of the display.

scopeCoords `ccIPair scopeCoords(const ccPoint &tabletP) const;`

Returns a point in scope coordinates for a tablet point you provide. This conversion is useful in drag animation.

Parameters

tabletP A tablet point you provide.

screenCoords `ccIPair screenCoords(const ccPoint &tabletP) const;`

Returns a point in screen coordinates for a tablet point you provide. This conversion is useful in drag animation.

Parameters

tabletP A tablet point you provide.

tabletCoords `ccPoint tabletCoords(const ccIPair &scopeP) const;`

Returns a point in tablet coordinates for a scope point you provide. This conversion is useful in drag animation.

Parameters

scopeP A scope point you provide.

visible

```
bool visible() const;
bool visible(bool vis);
```

- `bool visible() const;`

  Returns true if this tablet is visible. Returns false if it is not visible.

- `bool visible(bool vis);`

  When *vis* is set true this tablet is made visible. When *vis* is set false, this tablet is made invisible.

  Returns true if the visible state changed as a result of this call.

  Parameters

  vis The new visible flag, true or false.

Notes

Calling this function may cause a full screen refresh or a display erase.

This function should not be called on a default-constructed tablet used for recording.

lockScreenUpdatesPush

```
void lockScreenUpdatesPush(
  bool lock,
  Layers layer = eUnspecifiedLayer);
```

Increments or decrements the lock level to lock or unlock screen layer updates. When a layer is locked you can change its contents and the screen is not updated (refreshed). When the layer is unlocked the screen is updated each time you write into the layer. Locking updates while you are making layer changes can save time in your application by eliminating unnecessary screen updates.

Initially the lock level is 1 and screen updates are enabled. Calling **lockScreenUpdatesPush(***true***)** increments the lock level and **lockScreenUpdatesPop()** decrements the lock level. When the lock level is greater than 1, screen updates are disabled. When the lock level is decremented from 2 to 1 the screen is updated. Decrementing the lock level to less than 1 causes an error.

Parameters

lock	Lock the layer (true), or unlock the layer (false).
layer	The specified layer. *eUnspecifiedLayer* = all layers.

Throws

ccUIError::FlagStackUnderflow
: If the lock level is decremented below 1. The lock level is implemented using a flag stack. (See *flagstck.h*).

ccUIError::FlagStackOverflow
: If the lock level exceeds the upper limit (32). The lock level is implemented using a flag stack. (See *flagstck.h*).

ccUIError::BadLayer
: If *layer* is invalid.

You typically use **lockScreenUpdatesPush()** to lock screen updates and then use **lockScreenUpdatesPop()** to unlock screen updates and to refresh the screen. For example:

```
lockScreenUpdatesPush(true, ccUITablet::eUnspecifiedLayer);
```

```
// Do work here that would cause the display
// content to change (for example, drawing graphics)
// ...
// The display screen won't actually be updated until
// the following function call.

lockScreenUpdatesPop();
```

Notes

lockScreenUpdatesPush(*false***)** also decrements the lock level, however this is seldom used.

The lock level is implemented using a flag stack. (See *flagstck.h*).

lockScreenUpdatesPop

```
void lockScreenUpdatesPop(
  Layers layer = eUnspecifiedLayer);
```

Decrements the lock level to unlock screen layer updates. You typically use **lockScreenUpdatesPush(***true***)** to lock screen updates and then use **lockScreenUpdatesPop()** to unlock screen updates and to refresh the screen. See **lockScreenUpdatesPush()** above for more information.

Parameters

layer The specified layer. *eUnspecifiedLayer* = all layers.

Throws

ccUIError::FlagStackUnderflow
If the lock level is decremented below 1. The lock level is implemented using a flag stack. (See *flagstck.h*).

ccUIError::BadLayer
If *layer* is invalid.

screenUpdatesLocked

```
bool screenUpdatesLocked(Layers layer) const;
```

Returns true if the specified layer is locked. Returns false if it is not locked. If *layer* = *eUnspecifiedLayer*, all layers are specified.

See **lockScreenUpdatesPush()** and **lockScreenUpdatesPop()** above for information about locking screen updates.

Parameters

layer The specified layer. *eUnspecifiedLayer* = all layers.

Throws

ccUIError::BadLayer
If *layer* is invalid.

draw

```
void draw(const ccShape & s, const ccGraphicProps & props,
 Layers layer = eImageLayer);

void draw(const ccPoint& p, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const cc2Point& p, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccLineSeg& l, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccLine& l, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccCross& c, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccRect& r, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccCircle& c, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccAnnulus& a, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccEllipse2& e, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccEllipseArc2& e,
 const ccGraphicProps& props, Layers layer = eImageLayer);

void draw(const ccGenRect& r, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccGenAnnulus& a,
 const ccGraphicProps& props, Layers layer = eImageLayer);

void draw(const ccCoordAxes& a,
 const ccGraphicProps& props, Layers layer = eImageLayer);

void draw(const ccPointSet& p, const ccGraphicProps& props,
 Layers layer = eImageLayer);

void draw(const ccAffineRectangle& r,
 const ccGraphicProps& props, Layers layer = eImageLayer);

void draw(const ccPolyline & p,
 const ccGraphicProps & props, Layers layer = eImageLayer);

void draw(const ccBezierCurve & b,
```

```
  const ccGraphicProps & props, Layers layer = eImageLayer);

void draw(const ccCubicSpline & s,
  const ccGraphicProps & props, Layers layer = eImageLayer);

void draw(const ccEllipseAnnulusSection& eas,
  const ccGraphicProps& props, Layers layer = eImageLayer);

void draw(const ccGenPoly& gp, const ccGraphicProps& props,
  Layers layer = eImageLayer);

void draw(const cc2Wireframe& wf,
  const ccGraphicProps& props, Layers layer = eImageLayer);

void draw(const ccCvlString& s, const ccPoint& p,
  const ccColor& ink, const ccColor& backgr,
  const ccUIFormat& fmt, Layers layer = eImageLayer);

void draw(const ccPelBuffer_const<c_UInt8> &pb,
  const ccPoint& p, bool passThrough = false,
  c_UInt8 passColor = 0, bool displayRaw = true,
  Layers layer = eImageLayer);

void draw(const ccPelBuffer_const<c_UInt16> &pb,
  const ccPoint& p, bool passThrough = false,
  c_UInt16 passColor = 0, Layers layer = eImageLayer);
```

Note: This method is not supported.

```
void draw (const ccPelBuffer_const<ccPackedRGB16Pel> &pb,
  const ccPoint& p, bool passThrough = false,
  c_UInt16 passColor = 0, Layers layer = eImageLayer);

void draw (const ccPelBuffer_const<ccPackedRGB32Pel> &pb,
  const ccPoint& p, bool passThrough = false,
  c_UInt32 passColor = 0, Layers layer = eImageLayer);

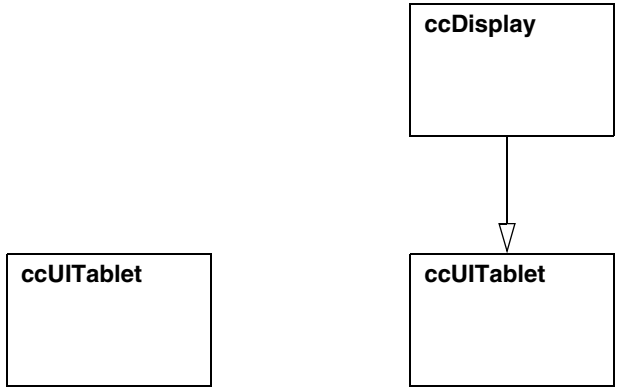
void draw(const ccRLEBuffer &rle, const ccPoint& p,
  const cmStd vector<c_UInt8>& colorMap,
  bool passThrough = true, c_UInt8 passColor = 0,
  bool ignoreScale = false, Layers layer = eImageLayer);

void draw (const ccUISketch& s, const ccPoint& p,
  Layers layer = eImageLayer);
```

The following notes apply to the draw graphics overloads, the first 22 **draw()** overloads described below. These overloads are the drawing functions for geometric shapes. They can record a static graphic into the tablet and/or render the graphic to a display

when called from a derived display class. Graphics must not contain degenerate geometry. For example, a rectangle cannot have a height or width of 0. The **ccPointSet** size must be greater than one.

It is important to understand the two ways these draw functions are used. See the following diagram:



Tablet
(For recording only)

Tablet + display
(For display and/or recording)

When you create a **ccUITablet** object and call its draw functions you can only record the drawings into its internal sketch list. There is no display support. When you program a **ccUITablet** object in this way you typically do the following:

1. Call the draw functions to record graphics into the tablet sketch list.
2. Call **ccUITablet::sketch()** to retrieve the sketch list.
3. Pass the retrieved sketch list to **ccDisplay::drawSketch()** to display the graphics.

Recording graphics into a tablet does not use the tablet's scaled map or base map. Recording occurs without regard to any offset or scale, it simply records the geometry given. When you program a display class such as **ccDisplay** that derives from **ccUITablet**, you typically call the draw functions to display graphics directly.

Notes

If you want static graphics to be a permanent part of the display, and to be automatically re-rendered when the viewport changes or a new image is inserted into the display, you should use **ccUITablet** derived classes that implement and support this functionality. (For example, **ccDisplay::drawSketch()**).

There is an implicit conversion from a **ccColor** object to a **ccGraphicProps** object, but the converse is not true.

There are some implicit conversions between some of the graphic types listed in ch_cvl/shapes.h. As such, there may not be a corresponding draw function for every shape (For example, **ccAnnulus** is implicitly converted to a **ccGenAnnulus**).

Not all graphical properties are supported for all graphical shapes. See the shape class documentation for which graphical properties are supported. For example, closed shape filling may not be fully supported for all shapes.

- ```
  void draw(const ccShape & s, const ccGraphicProps & props,
    Layers layer = eImageLayer);
  ```

  Draws a shape in the specified layer using the specified graphic properties.

  Parameters

s	The shape.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- ```
  void draw(const ccPoint& p, const ccGraphicProps& props,
    Layers layer = eImageLayer);
  ```

  Draws a point in the specified layer using the specified graphic properties.

  Parameters

p	The point.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- `void draw(const cc2Point& p, const ccGraphicProps& props, Layers layer = eImageLayer);`

Draws a point in the specified layer using the specified graphic properties. This method draws a **cc2Point** which derives from **ccShape**. To draw a **ccPoint** use the draw method above.

Parameters

p The point.

props The graphic properties of the shape.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- `void draw(const ccLineSeg& l, const ccGraphicProps& props, Layers layer = eImageLayer);`

Draws a line segment in the specified layer using the specified graphic properties.

Parameters

l The line segment.

props The graphic properties of the shape.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- `void draw(const ccLine& l, const ccGraphicProps& props, Layers layer = eImageLayer);`

Draws a line in the specified layer using the specified graphic properties.

Parameters

l	The line segment.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccCross& c, const ccGraphicProps& props, Layers layer = eImageLayer);`

Draws a cross in the specified layer using the specified graphic properties.

Parameters

c	The cross.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccRect& r, const ccGraphicProps& props, Layers layer = eImageLayer);`

Draws a rectangle in the specified layer using the specified graphic properties.

Parameters

r	The rectangle.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccCircle& c, const ccGraphicProps& props, Layers layer = eImageLayer);`

  Draws a circle in the specified layer using the specified graphic properties.

  Parameters

c	The circle.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccAnnulus& a, const ccGraphicProps& props, Layers layer = eImageLayer);`

  Draws an annulus in the specified layer using the specified graphic properties.

  Parameters

a	The annulus.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccEllipse2& e, const ccGraphicProps& props, Layers layer = eImageLayer);`

  Draws an ellipse in the specified layer using the specified graphic properties.

  Parameters

e	The ellipse.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- ```
  void draw(const ccEllipseArc2& e,
    const ccGraphicProps& props, Layers layer = eImageLayer);
  ```

  Draws an ellipse arc in the specified layer using the specified graphic properties.

  Parameters

  e The ellipse arc.

  props The graphic properties of the shape.

  layer The layer to draw into. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

- ```
  void draw(const ccGenRect& r, const ccGraphicProps& props,
    Layers layer = eImageLayer);
  ```

  Draws a generalized rectangle in the specified layer using the specified graphic properties.

  Parameters

  r The generalized rectangle.

  props The graphic properties of the shape.

  layer The layer to draw into. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

- ```
  void draw(const ccGenAnnulus& a,
    const ccGraphicProps& props, Layers layer = eImageLayer);
  ```

  Draws a generalized annulus in the specified layer using the specified graphic properties.
  ```

Parameters

a	The generalized annulus.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- ```
  void draw(const ccCoordAxes& a,
    const ccGraphicProps& props, Layers layer = eImageLayer);
  ```

  Draws a set of coordinate axes in the specified layer using the specified graphic properties.

  Parameters

a	The coordinate axes.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- ```
  void draw(const ccPointSet& p, const ccGraphicProps& props,
    Layers layer = eImageLayer);
  ```

  Draws a point set in the specified layer using the specified graphic properties.

  Parameters

p	The point set. A point set size must be greater than 1.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccAffineRectangle& r, const ccGraphicProps& props, Layers layer = eImageLayer);`

  Draws an affine rectangle in the specified layer using the specified graphic properties.

  Parameters

r	The affine rectangle.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccPolyline & p, const ccGraphicProps & props, Layers layer = eImageLayer);`

  Draws a polyline in the specified layer using the specified graphic properties.

  Parameters

p	The polyline.
props	The graphic properties of the polyline.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void draw(const ccBezierCurve & b, const ccGraphicProps & props, Layers layer = eImageLayer);`

  Draws a bezier curve in the specified layer using the specified graphic properties.

  Parameters

b	The bezier curve.
props	The graphic properties of the shape.
layer	The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- `void draw(const ccCubicSpline & s, const ccGraphicProps & props, Layers layer = eImageLayer);`

  Draws a cubic spline in the specified layer using the specified graphic properties.

  Parameters

  s The cubic spline.

  props The graphic properties of the shape.

  layer The layer to draw into. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

- `void draw(const ccEllipseAnnulusSection& eas, const ccGraphicProps& props, Layers layer = eImageLayer);`

  Draws an ellipse annulus in the specified layer using the specified graphic properties.

  Parameters

  eas The ellipse annulus section.

  props The graphic properties of the shape.

  layer The layer to draw into. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

- `void draw(const ccGenPoly& gp, const ccGraphicProps& props, Layers layer = eImageLayer);`

  Draws a generalized polygon in the specified layer using the specified graphic properties.

Parameters

gp	The generalized polygon.
props	The drawing properties.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

Notes

To show or hide vertices for a generalized polygon, use the **ccGraphicProps::showVertex()** method.

- ```
  void draw(const cc2Wireframe& wf,
    const ccGraphicProps& props, Layers layer = eImageLayer);
  ```

  Draws a two-dimensional shape in the specified layer using the specified graphic properties.

  Parameters

wf	The wireframe shape to draw.
props	The drawing properties.
layer	The layer in which to draw.

  Notes

  The direction (edge polarity) arrows are drawn if and only if the **arrowHead()** field of *props* is true. Sharp vertex corners are superimposed over rounded ones if and only if the **showVertex()** field of *props* is true.

- ```
  void draw(const ccCvlString& s, const ccPoint& p,
    const ccColor& ink, const ccColor& backgr,
    const ccUIFormat& fmt, Layers layer = eImageLayer);
  ```

  Draws a character string starting at the specified point, in the specified color and format, with the specified background, in the specified layer.

  Notes

  This draw string function behaves the same as the draw graphics functions above and follows the same rules for recording and displaying. See the description on page 3356.

Parameters

s	The text string to draw.
p	The location of the upper left corner of the string. *p* is in the coordinate space specified when adding the text to a display using **ccDisplay::drawSketch()**.
ink	The color of the text.
backgr	The background color. This is the area within the bounding rectangle of the text but not including the text itself.
fmt	The text format contains the font and alignment. The alignment location is positioned at point *p*.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- ```
  void draw(const ccPelBuffer_const<c_UInt8> &pb,
    const ccPoint& p, bool passThrough = false,
    c_UInt8 passColor = 0, bool displayRaw = true,
    Layers layer = eImageLayer);
  ```

  Draws a grey-scale pel buffer.

  Parameters

pb	The grey scale image to draw.
p	The point at which to draw the pel buffer. *p* is in the coordinate space specified when adding *pb* to a display using **ccDisplay::drawSketch()**.
passThrough	If true, pixels with the value *passColor* are not drawn.
passColor	The pass-through color. If *passThrough* is true, pixels with value *passColor* are not drawn.
displayRaw	If false, values in the image are mapped to the range *ccUIRootObject::eFirstUserColor* through *ccUIRootObject::eFirstUIReservedColor* - 1 before being displayed.
layer	The layer to draw into. Must be one of:
  ```

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

Notes

The pixel map is applied before the *passColor* comparison is performed.

This draw pel buffer function behaves the same as the draw graphics functions above and follows the same rules for recording and displaying. See the description on page 3356.

- ```
  void draw(const ccPelBuffer_const<c_UInt16> &pb,
    const ccPoint& p, bool passThrough = false,
    c_UInt16 passColor = 0, Layers layer = eImageLayer);
  ```

Draws a 16-bit grey-scale pel buffer.

Notes

This method is not supported.

- ```
  void draw(const ccPelBuffer_const<ccPackedRGB16Pel> &pb,
    const ccPoint& p, bool passThrough = false,
    c_UInt16 passColor = 0, Layers layer = eImageLayer);
  ```

Draws a color (RGB 5-6-5) pel buffer to the display. If *passThrough* is true, pixels with value *passColor* are not drawn.

Calling this method with recording on is not supported. You cannot use this method to add a **ccPackedRGB16Pel** pel buffer into a recording tablet. That operation has no effect. For a discussion of the draw function rules for recording and displaying see the description on page 3356.

Parameters

pb	The pel buffer to draw.
p	The point at which to draw the pel buffer. *p* is in the coordinate space specified when adding *pb* to a display using **ccDisplay::drawSketch()**.
passThrough	If true, pixels with the value *passColor* are not drawn.
passColor	The pass-through color. If *passThrough* is true, pixels with value *passColor* are not drawn.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

Notes

To permanently display this type of pel buffer, see the **ccDisplay::image()** overload that takes the same argument type.

- ```
  void draw(const ccPelBuffer_const<ccPackedRGB32Pel> &pb,
    const ccPoint& p, bool passThrough = false,
    c_UInt32 passColor = 0, Layers layer = eImageLayer);
  ```

Draws a 32-bit (RGB 0-8-8-8)pel buffer. If *passThrough* is true, pixels with value *passColor* are not drawn.

Calling this method with recording on is not supported. You cannot use this method to add a **ccPackedRGB32Pel** pel buffer into a recording tablet. That operation has no effect. For a discussion of the draw function rules for recording and displaying see the description on page 3356.

Parameters

pb The pel buffer to draw.

p The point at which to draw the pel buffer. *p* is in the coordinate space specified when adding *pb* to a display using **ccDisplay::drawSketch()**.

passThrough If true, pixels with the value *passColor* are not drawn.

passColor The pass-through color. If *passThrough* is true, pixels with value *passColor* are not drawn.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

Notes

To permanently display this type of pel buffer, see the **ccDisplay::image()** overload that takes the same argument type.
  ```

- ```
  void draw(const ccRLEBuffer &rle, const ccPoint& p,
    const cmStd vector<c_UInt8>& colorMap,
    bool passThrough = true, c_UInt8 passColor = 0,
    bool ignoreScale = false, Layers layer = eImageLayer);
  ```

  Draws a run-length encoded image using the specified color map.

  Parameters

rle	The run-length encoded image to draw.
p	The point at which to draw the RLE buffer. Specified in the coordinate space used when adding the RLE buffer to a display using **ccDisplay::drawSketch()**.
colorMap	A vector of up to 256 values of type **c_UInt8**, each representing an index into the color map returned by **ccDisplay::colorMap()**. The size of the color map vector depends on the number of grey levels in the RLE buffer to be drawn.
passThrough	If true, pixels with the value *passColor* are not drawn.
passColor	The pass-through color. If *passThrough* is true, pixels with value *passColor* are not drawn.
ignoreScale	If true, the image is not expanded or subsampled. This value is used when drawing icons.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

  Notes

  You typically initialize the color map using predefined instances of **ccColor** and the **ccColor::index()** method. For example, to create a color map with two elements and initialize them to the pass-through color and red, use the following code:

  ```
  cmStd vector<c_UInt8> colorMap(2);
  colorMap[0] = ccColor::passColor().index();
  colorMap[1] = ccColor::redColor().index();
  ```

  The color map is applied before the pass-through comparison is made. The pass-through color used for RLE buffers (index value 0) is different from the pass-through color used for general display (index value 243).

  See the *Color Maps* section of the *Displaying Images* chapter in the *CVL User's Guide* for more information on color maps.
  ```

To permanently add the RLE buffer to a display, this function must be called from a tablet with recording enabled, and then the tablet's sketch must be added to the display. For a description of how to use the draw functions to record and display RLE buffers, see the discussion on page 3356.

- ```
  void draw (const ccUISketch& s, const ccPoint& p,
    Layers layer = eImageLayer);
  ```

Displays the recorded sketch *s*. All coordinates in the sketch are relative to the specified point. The specified layer is used only for sketch components for which no layer is specified. If both *layer* and the recorded sketch specify *eUnspecifiedLayer*, the sketch is not displayed.

For tablet objects with no display, it records the sketch *s* into this tablet's sketch.

Parameters

s	The sketch.
p	The orientation point of the sketch. All coordinates in the sketch are relative to this point.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346. If both the *layer* argument and the sketch component use *eUnspecifiedLayer*, the component is not drawn.

Notes

Do not pass in this tablet's **sketch()** as the first argument. The intent is for other tablets to call this function to record their sketch list and/or display their sketch. The following construct is not allowed: **tablet.draw(tablet.sketch())**.

textRect

```
ccUIRect textRect (const ccCvlString& s, const ccPoint& p,
  const ccUIFormat& fmt, ccUIRect& clippedRect,
  Layers layer=eImageLayer);
```

Returns the rectangle that would enclose the string *s* if it were drawn. If the string is clipped by an area of the display, then the clipped part of the rectangle is returned in *clippedRect*. Otherwise, *clippedRect* is the same as the returned rectangle.

Parameters

s	The text string to be measured.
```

p	The upper left corner of the string. This is the offset in the current tablet coordinate space.
fmt	The text format. Contains alignment and font information.
clippedRect	The clipped rectangle if the string needed to be clipped. Otherwise it is the same as the returned **ccUIRect**.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

Notes

This method is intended to be called from a display class derived from **ccUITablet**. Do not call this function from a default constructed **ccUITablet**.

When calling this function to get the text rectangle of an interactive **ccUILabel** the offset point *p* and returned values are in the coordinate space to which the interactive graphic was added to the display from **ccDisplay::addShape()**. For example:

```
ccUILabel *uiLabel = new ccUILabel;
myDisplay.addShape(uiLabel, ccDisplay::eDisplayCoords);
uiLabel->tablet()->textRect(...).
```

When calling this method on a text string from a display that contains an image, the offset *p* and returned values are in client coordinates.

This function is not intended to be called from a tablet constructed for recording. It returns a NULL rectangle when called on such a tablet.

drawPointIcon

```
void drawPointIcon (
  const ccPoint& p,
  const ccGraphicProps& props,
  Layers layer=eImageLayer);
```

Draws a static point icon centered at the specified location.

Parameters

p	The point at which to draws the point icon.
props	The drawing properties of the point icon.
layer	The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

Notes

This function behaves the same as the draw graphics functions above and follows the same rules for recording and displaying. See the description on page 3356.

set

```
void set(const ccColor& color, Layers layer=eImageLayer);

void set(const ccUIRect& rect, const ccColor& color,
  Layers layer=eImageLayer);

void set(const ccRect& rect, const ccColor& color,
  Layers layer=eImageLayer);
```

Set all or part of the specified layer to the specified color.

When calling this function on a display class derived from **ccUITablet**, the operation takes effect immediately as long as there is a scope associated with the tablet, but is *temporary* unless this operation is part of a tablet sketch.

Regarding temporary operations, this function behaves the same as the draw graphics functions and follows the same rules for recording and displaying. See the description on page 3356.

The tablet base map transform must not contain rotation or skew when using this method. This method doesn't draw filled or rotated rectangles.

- `void set(const ccColor& color, Layers layer=eImageLayer);`

Set the entire specified layer to the specified color. If recording is enabled, all previously recorded components in the specified layer are erased.

This version of **set()** uses scope coordinates.

Parameters

color The new color of the layer.

layer The layer. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- `void set(const ccUIRect& rect, const ccColor& color, Layers layer=eImageLayer);`

  Set the area described by *rect* in the specified layer, to the specified color. This version of **set()** uses tablet coordinates.

  Parameters

rect	The area whose color to set.
color	The new color.
layer	The layer to change. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- `void set(const ccRect& rect, const ccColor& color, Layers layer=eImageLayer);`

  Set the area described by *rect* in the specified layer, to the specified color. This version of **set()** uses scope coordinates.

  Parameters

rect	The area whose color to set.
color	The color of the layer.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

setComplement

```
void setComplement (const ccUIRect& rect,
 const ccColor& color, Layers layer=eImageLayer);

void setComplement (const ccRect& rect,
 const ccColor& color, Layers layer=eImageLayer);
```

Sets all of the specified layer to the specified color except for the area enclosed by *rect*. The area enclosed by *rect* is unchanged.

When calling this function on a display class derived from **ccUITablet**, the operation takes effect immediately as long as there is a scope associated with the tablet, but is *temporary* unless this operation is part of a tablet sketch.

Regarding temporary operations, this function behaves the same as the draw graphics functions and follows the same rules for recording and displaying. See the description on page 3356.

The tablet **baseMap()** transform must not contain rotation or skew when using this method. This method doesn't draw filled or rotated rectangles.

- ```
  void setComplement (const ccUIRect& rect,
   const ccColor& color, Layers layer=eImageLayer);
  ```

  This version of **setComplement()** uses the scope coordinate system regardless of the coordinate system you use to draw the tablet's sketch.

  Parameters

rect	The area whose color is unchanged.
color	The new color.
layer	The layer to color. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

- ```
  void setComplement (const ccRect& rect,
   const ccColor& color, Layers layer=eImageLayer);
  ```

  This version of **setComplement()** uses the tablet coordinate system regardless of the coordinate system you use to draw the tablet's sketch.

  Parameters

rect	The area whose color is unchanged.

color The new color.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

fill

```
void fill (const ccIPair& pt, const ccColor& fillClr,
  const ccColor& clr, FillType fillType,
  Layers layer=eImageLayer);

void fill (const ccPoint& pt, const ccColor& fillClr,
  const ccColor& clr, FillType fillType,
  Layers layer=eImageLayer);
```

Fills in an area with a specified color.

If *fillType* is *ccUITablet::eFillBorder*, this function starts filling from the point *pt* in all directions until it reaches the border color *clr*. If *fillType* is *ccUITablet::eFillSurface*, filling starts at *pt* and continues in all directions until the color is not *clr*.

When calling this function on a display class derived from **ccUITablet**, the operation takes effect immediately as long as there is a scope associated with the tablet, but is *temporary* unless this operation is part of a tablet sketch.

Regarding temporary operations, this function behaves the same as the draw graphics functions and follows the same rules for recording and displaying. See the description on page 3356.

- ```
  void fill (const ccIPair& pt, const ccColor& fillClr,
    const ccColor& clr, FillType fillType,
    Layers layer=eImageLayer);
  ```

  This version of **fill()** always uses the scope coordinate system regardless of the coordinate system you use to draw the tablet's sketch.

  Parameters

  pt The point at which to start filling.

  fillClr The color to use for filling.

  clr The *stop* color.

  fillType *ccUITable::eFillBorder*: fills until the stop color is encountered.
  ```

ccUITable::eFillSurface: fills until a color other than the stop color is encountered.

layer — The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- ```
  void fill (const ccPoint& pt, const ccColor& fillClr,
    const ccColor& clr, FillType fillType,
    Layers layer=eImageLayer);
  ```

This version of **fill()** uses the coordinate system that you use to draw the tablet's sketch.

Parameters

pt — The point at which to start filling.

fillClr — The color to use for filling.

clr — The *stop* color.

fillType — *ccUITable::eFillBorder*: fills until the stop color is encountered.

ccUITable::eFillSurface: fills until a color other than the stop color is encountered.

layer — The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.
  ```

drawStart

```
void drawStart (const ccPoint& pt,
  Layers layer=eImageLayer);
```

Sets the *drawing point* in the specified layer. The drawing point is used as the current pen location by some tablet drawing routines. Drawing routines never change the drawing point, unless otherwise specified.

When calling this function on a display class derived from **ccUITablet**, the operation takes effect immediately as long as there is a scope associated with the tablet, but is *temporary* unless this operation is part of a tablet sketch.

Regarding temporary operations, this function behaves the same as the draw graphics functions and follows the same rules for recording and displaying. See the description on page 3356.

Parameters

pt	The initial location of the drawing point.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

drawTo

```
void drawTo(
  const ccPoint& pt,
  const ccGraphicProps& props,
  Layers layer=eImageLayer);
```

Draws a line segment from the current drawing point to a point *pt*. Updates the drawing point to the point *pt* after the line is drawn.

When calling this function on a display class derived from **ccUITablet**, the operation takes effect immediately as long as there is a scope associated with the tablet, but is *temporary* unless this operation is part of a tablet sketch.

Regarding temporary operations, this function behaves the same as the draw graphics functions and follows the same rules for recording and displaying. See the description on page 3356.

Parameters

pt	The line segment end point.
props	The graphic properties of the line segment.
layer	The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

moveToRelPels

```
void moveToRelPels (c_Int32 x, c_Int32 y,
  Layers layer=eImageLayer);

void moveToRelPels (const cc2Vect& tabletDir,
  float distPels, Layers layer=eImageLayer);
```

Move the current drawing point and update it. No drawing is performed.

When calling this function on a display class derived from **ccUITablet**, the operation takes effect immediately as long as there is a scope associated with the tablet, but is *temporary* unless this operation is part of a tablet sketch.

Regarding temporary operations, this function behaves the same as the draw graphics functions and follows the same rules for recording and displaying. See the description on page 3356.

- ```
  void moveToRelPels (c_Int32 x, c_Int32 y,
    Layers layer=eImageLayer);
  ```

  Moves the drawing point to a point that is *x* and *y* pixels away from the current drawing point.

  Parameters

x	The number of pixels in the x-direction to add to the current drawing point.
y	The number of pixels in the y-direction to add to the current drawing point.
layer	The specified layer. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.
  ```

- ```
  void moveToRelPels (const cc2Vect& tabletDir,
    float distPels, Layers layer=eImageLayer);
  ```

  Moves the drawing point to a point that is *dist* pixels away frmo the current drawing point, in the direction specified by the vector *tabletDir*. The direction uses the coordinate system used to draw the tablet's sketch.

  Parameters

tabletDir	The direction in which to draw the line.
distPels	The length of the line segment to draw in scope coordinates.
layer	The specified layer. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

  Notes

  The length of the direction vector is ignored.

drawToRelPels

```
void drawToRelPels(
  c_Int32 x,
  c_Int32 y,
  const ccGraphicProps& props,
  Layers layer = eImageLayer);

void drawToRelPels(
  const cc2Vect& tabletDir,
  float distPels,
  const ccGraphicProps& props,
  Layers layer = eImageLayer);
```

- ```
  void drawToRelPels(
    c_Int32 x,
    c_Int32 y,
    const ccGraphicProps& props,
    Layers layer = eImageLayer);
  ```

  Draws a line segment from the current drawing point to a point that is *x* and *y* pixels away from it, and then updates the current drawing point.

  Parameters

x	The number of pixels in the x-direction to add to the current drawing point.

y The number of pixels in the y-direction to add to the current drawing point.

props The drawing properties.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- ```
  void drawToRelPels(
    const cc2Vect& tabletDir,
    float distPels,
    const ccGraphicProps& props,
    Layers layer = eImageLayer);
  ```

  Draws a line segment from the current drawing point to a point *distPels* from the drawing point in the direction specified by the vector *tabletDir*. The direction uses the coordinate system you use to draw the tablet's sketch. Updates the current drawing point.

  Parameters

  tabletDir The direction in which to draw the line.

  distPels The length, in pixels, of the line segment to draw.

  props The drawing properties.

  layer The layer to draw into. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

  Notes

  The length of the direction vector is ignored.

drawingPoint

```
ccPoint drawingPoint(Layers layer=eImageLayer) const;
```

Returns the current drawing point using the coordinate system that you use to draw the tablet's sketch.

Parameters

layer The specified layer. Must be one of:
```

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

Throws

ccUIError::BadLayer
If *layer* is invalid.

drawPointArg

```
const ccPoint& drawPointArg () const;
```

Returns a special **ccPoint** that can be used as an argument to any drawing routine to specify the current drawing point.

outline

```
void outline(const ccUIRect& rect, const ccColor& color,
  c_Int32 thickness, Layers layer=eImageLayer);

void outline(const ccRect& rect, const ccColor& color,
  c_Int32 thickness, Layers layer=eImageLayer);
```

Draws a rectangular outline into the specified layer. The tablet **baseMap()** transform must not contain rotation or skew when using these methods. The methods do not draw filled or rotated rectangles.

-
```
void outline(const ccUIRect& rect, const ccColor& color,
  c_Int32 thickness, Layers layer=eImageLayer);
```

Draws an outline of *rect* in the specified color and thickness. This version of **outline()** uses the scope coordinate system regardless of the coordinate system you use to draw the tablet's sketch.

Parameters

rect The rectangle whose outline is to be drawn.

color The color to use for drawing.

thickness The thickness of the outline in pixels.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- `void outline(const ccRect& rect, const ccColor& color, c_Int32 thickness, Layers layer=eImageLayer);`

  Draws an outline of *rect* in the specified color and thickness. This version of **outline()** uses the coordinate system that you use to draw the tablet's sketch.

  Parameters

rect	The rectangle whose outline is to be drawn.
color	The color to use for drawing.
thickness	The thickness of the outline in pixels.
layer	The layer to draw into. Must be one of: *ccUITablet::eImageLayer* *ccUITablet::eOverlayLayer* *ccUITablet::eUnspecifiedLayer* See *Layers* on page 3346.

outline3D

```
void outline3D(const ccUIRect& rect, const ccColor& light,
 const ccColor& shadow, c_Int32 thickness,
 Layers layer=eImageLayer);

void outline3D(const ccRect& rect, const ccColor& light,
 const ccColor& shadow, c_Int32 thickness,
 Layers layer=eImageLayer);
```

Draws a rectangular outline into the specified layer. The rectangle is drawn with highlights and shadows for a 3D effect. The tablet **baseMap()** transform must not contain rotation or skew when using these methods. The methods do not draw filled or rotated rectangles.

- `void outline3D(const ccUIRect& rect, const ccColor& light, const ccColor& shadow, c_Int32 thickness, Layers layer=eImageLayer);`

  Draws an outline of *rect* in the specified thickness using the *light* color for the top and left edges and the *shadow* color for the bottom and right edges. This version of **outline3D()** always uses the scope coordinate system regardless of the coordinate system you use to draw the tablet's sketch.

  Parameters

rect	The rectangle whose outline is to be drawn.
light	The color to use for drawing the left and top edges.

shadow The color to use for drawing the right and bottom edges.

thickness The thickness of the outline in pixels.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- ```
  void outline3D(const ccRect& rect, const ccColor& light,
    const ccColor& shadow, c_Int32 thickness,
    Layers layer=eImageLayer);
  ```

  Draws an outline of *rect* in the specified thickness using the *light* color for the top and left edges and the *shadow* color for the bottom and right edges. This version of **outline()** uses the coordinate system that you use to draw the tablet's sketch.

  Parameters

  rect The rectangle whose outline is to be drawn.

  light The color to use for drawing the left and top edges.

  shadow The color to use for drawing the right and bottom edges.

  thickness The thickness of the outline in pixels.

  layer The layer to draw into. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

arrowHead

```
void arrowHead (const ccPoint& pt, const cc2Vect& dir,
  float armLen, const ccGraphicProps& props,
  Layers layer=eImageLayer);
```

Draws an arrowhead pointing in the direction *dir* with its vertex at the point *pt*. Updates the current drawing point to *pt*.

Parameters

pt The vertex of the arrowhead in the coordinate system that you use to draw the tablet's sketch.

dir The direction that the arrowhead points.
```

armLen The length of the arrowhead's arms in pixels.

props The drawing properties.

layer The layer to draw into. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

Notes

The length of the vector is ignored.

erase

```
void erase(Layers layer = eUnspecifiedLayer);

void erase(
  const ccUIRect& rect,
  Layers layer = eUnspecifiedLayer);
```

Erases all or the selected part (*rect*, in scope coords) of this tablet. The layer you specify causes the following:

Specified layer	Description
eImageLayer	Erases the image layer
eOverlayLayer	Erases the overlay layer
eUnspecifiedLayer	Erases both the image layer and the overlay layer

This function erases but does not redraw the display. You will not see the erase effect until you do a redraw.

- `void erase(Layers layer = eUnspecifiedLayer);`

Erases the entire sketch of this tablet. See the above description.

Parameters

layer The layer to erase. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

- ```
  void erase(
    const ccUIRect& rect,
    Layers layer = eUnspecifiedLayer);
  ```

  Erases the area enclosed by *rect*, of this tablet's sketch. See the above description.

  Parameters

  rect The portion of the sketch to erase in scope coordinates.

  layer The layer to erase. Must be one of:

  ccUITablet::eImageLayer
  ccUITablet::eOverlayLayer
  ccUITablet::eUnspecifiedLayer

  See *Layers* on page 3346.

fullDraw

```
void fullDraw(Layers layer = eUnspecifiedLayer);
```

Redraws this tablet and all other tablets managed by the display system, in back-to-front order.

Parameters

layer The layer to draw. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.

redraw

```
void redraw(Layers = eUnspecifiedLayer);
```

Redraws this tablet. This method temporarily disables recording during the redraw operation.

Parameters

Layers The layer to redraw. Must be one of:

ccUITablet::eImageLayer
ccUITablet::eOverlayLayer
ccUITablet::eUnspecifiedLayer

See *Layers* on page 3346.
```

Static Functions

overlaySupported

```
static bool overlaySupported();
```

Returns true if the overlay layer is supported for your display.

Notes

Some hardware platforms do not support overlay graphics.

makeIcon

```
static ccRLEBuffer makeIcon(const TCHAR* pattern,
  const cmStd vector<c_UInt8>& colorMap);
```

This function creates an icon as a run-length encoded buffer using the specified pattern and color map. *pattern* must be non-NULL and point to a NULL-terminated string that is formed strictly according to the following rules.

The pattern is a string consisting of the digits 0 through 9, dot (.), plus (+), and a terminator (/, or newline). The first character of the string specifies the terminator (typically either "/" or newline). The terminator ends each row and all rows must be the same size. Other characters specify pixel values. The digits are used to lookup values in the *colorMap*. Space, dot, and plus refer to the pass value (**ccColor::passColor()**), Space and dot give you some artistic freedom. Plus is used when the pattern is also used to specify a touchMap (see **ccUIIconShape**).

Parameters

pattern	An icon pattern.
colorMap	The color map to map the values 0 through 9 into the range 0 through 256.

Example

```
const TCHAR* cross =
    "/"
    "...+0+.../"
    "...+0+.../"
    "...+0+.../"
    "++++0++++/"
    "000000000/"
    "++++0++++/"
```

```
    "...+0+.../"
    "...+0+.../"
    "...+0+.../";
```

Throws

ccUIError::BadIconPattern
The icon pattern does not conform to the rules.

iconTouched

```
static bool iconTouched(const TCHAR* touchMap,
  const ccIPair& p, c_Int32 touchMapWidth = -1);
```

Returns true if the icon specified by the pattern *touchMap* is touched at the point *p*. *p* is relative to the upper left corner of the icon. This function returns true if the pixel designated by *p* is a digit or a + in the map.

Parameters

touchMap	An icon pattern.
p	The point.
touchMapWidth	The width of the touch map. If –1, compute the touch map width from the pattern.

Deprecated Members

The following functions are provided for backward compatibility only and should not be used for new development. These functions may be removed and not supported in a future CVL release.

draw

The drawing functions for **ccEllipse** and **ccEllipseArc** have been replaced with functions that use **ccEllipse2** and **ccEllipseArc2**.

```
void draw(const ccEllipse & e, const ccGraphicProps & props,
Layers layer = eImageLayer);

void draw(const ccEllipseArc & e,
 const ccGraphicProps & props, Layers layer = eImageLayer);
```

The drawing function for **ccPolygon** has been replaced with the drawing function that uses **ccPolyline**.

```
void draw(const ccPolygon& p, const ccGraphicProps& props,
 Layers layer = eImageLayer);
```

The following functions that use `const ccColor&` have been replaced with drawing functions that use `const ccGraphicProps&`.

```
void draw(const ccPoint& p, const ccColor& color,
 Layers layer = eImageLayer);

void draw(const ccLineSeg& l, const ccColor& color,
 Layers layer = eImageLayer);

void draw(const ccLine& l, const ccColor& color,
 Layers layer = eImageLayer);

void draw(const ccCross& c, const ccColor& color,
 Layers layer = eImageLayer);

void draw(const ccRect& r, const ccColor& color,
 Layers layer = eImageLayer, bool fill = false);

void draw(const ccCircle& c, const ccColor& color,
 Layers layer = eImageLayer, bool fill = false);

void draw(const ccAnnulus& c, const ccColor& color,
 Layers layer = eImageLayer);

void draw(const ccEllipse& e, const ccColor& color,
 Layers layer = eImageLayer, bool fill = false);

void draw(const ccEllipseArc& e, const ccColor& color,
 Layers layer = eImageLayer);

void draw(const ccGenRect& r, const ccColor& color,
 Layers layer = eImageLayer, bool fill = false);

void draw(const ccGenAnnulus& a, const ccColor& color,
 Layers layer = eImageLayer);
```

```
void draw(const ccCoordAxes& a, const ccColor& color,
  Layers layer = eImageLayer);

void draw(const ccPointSet& p, const ccColor& color,
  Layers layer = eImageLayer);

void draw(const ccAffineRectangle& r, const ccColor& color,
  Layers layer = eImageLayer);

void draw(const ccPolygon& p, const ccColor& color,
  Layers layer = eImageLayer);

void draw(const ccEllipseAnnulusSection& eas,
  const ccColor& color, Layers layer = eImageLayer,
  bool drawArrowHead = true,
  bool drawArrowHeadForward = true);

void draw(const ccGenShape& gs, const ccColor& color,
  Layers layer = eImageLayer);

void draw(const ccGenPoly& gp, const ccColor& color,
  Layers layer = eImageLayer, bool showVertex = false);

void draw(const cc2Wireframe& wf,
  const ccColor& c, Layers layer = eImageLayer, bool
showVertex = false);

void draw(const class ccGraphic& g,
  Layers layer = eImageLayer);
```

drawPointIcon

```
void drawPointIcon (const ccPoint& p, const ccColor& color,
  Layers layer=eImageLayer);
```

drawTo

```
void drawTo(const ccPoint& p, const ccColor& color,
  Layers layer = eImageLayer);
```

drawToRelPels

```
void drawToRelPels(c_Int32 x, c_Int32 y,
  const ccColor& color, Layers layer = eImageLayer);

void drawToRelPels(const cc2Vect& tabletDir,
  float distPels, const ccColor& color,
  Layers layer = eImageLayer);
```

drawPixel

```
void drawPixel (const ccDPair& p, const ccColor& color,
  Layers layer=eImageLayer);
```

drawLineSeg

```
void drawLineSeg (const ccDPair& pt1, const ccDPair& pt2,
  const ccColor& color, Layers layer=eImageLayer);
```

drawLine

```
void drawLine (const ccDPair& pt, ccRadian t,
  const ccColor& color, Layers layer=eImageLayer);
```

drawCross

```
void drawCross (const ccDPair& center,
const ccRadian& angle, const ccColor& color,
Layers layer=eImageLayer);
```

drawRect

```
void drawRect (const ccDPair& ul, const ccDPair& lr,
  const ccColor& color, Layers= eImageLayer);
```

drawCircle

```
void drawCircle (const ccDPair& center, double radius,
  const ccColor& color, Layers layer=eImageLayer);
```

drawEllipse

```
void drawEllipse (const ccDPair& center,
  const ccDPair& radii, ccRadian t, const ccColor& color,
  Layers layer=eImageLayer);
```

drawEllipseArc

```
void drawEllipseArc (const ccDPair& center, const ccDPair&
radii, ccRadian t, ccRadian phi1, ccRadian phi2,
  const ccColor& color, Layers layer=eImageLayer);
```

drawGenRect

```
void drawGenRect (const ccDPair& center,
```

```
const ccDPair& radii, ccRadian& t, const ccDPair& round,
const ccColor& color, Layers layer=eImageLayer);
```

drawAffineRect

```
void drawAffineRect(const ccDPair& center, double xLen,
 double yLen, const ccRadian& xRotation,
 const ccRadian& skew, const ccColor& color,
 Layers layer=eImageLayer);
```

drawPolygon

```
void drawPolygon (const cmStd vector<cc2Vect>& pts,
 ccPolygon::PolyFillMode fm, const ccColor& color,
 Layers layer=eImageLayer);
```

scratchPad

```
static ccUITablet scratchPad;
```

A tablet in record mode with a NULL scope to make sketches on. Erase before use.

Notes

This variable has been deprecated. Use the default constructor to create a tablet for recording.

drawOn

```
ccFlagStack& drawOn();
```

recordOn

```
ccFlagStack& recordOn();
```

drawOnOverlay

```
ccFlagStack& drawOnOverlay();
```

ccUITablet

ccVersion

```
#include <ch_cvl/version.h>

class ccVersion;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

Constructors/Destructors

ccVersion

```
ccVersion(c_UInt32 version, c_UInt32 details,
  const ccCvlString& product = cmT("CVL"));
```

Constructs a version object with the given version information.

Parameters

version Hexadecimal number specifying the CVL version. See also **cmCvlVersion** on page 3396.

details Hexadecimal number specifying the version details. See also **cmCvlVersionDetails** on page 3397.

product The product name, set by default to “CVL”.

Enumerations

ReleaseType

```
enum ReleaseType;
```

Specifies the type of release.

Value	Meaning
eRelease = 0	Released software
ePreRelease	Prerelease software

Operators

operator== `bool operator==(const ccVersion& rhs) const;`

Returns true if this version is equal to *rhs*, false otherwise. Use this operator only after verifying that **ccVersion::canCompare(rhs)** is true.

Parameters

rhs The other version.

operator!= `bool operator!=(const ccVersion& rhs) const;`

Returns true if this version is not equal to *rhs*, false otherwise. Use this operator only after verifying that **ccVersion::canCompare(rhs)** is true.

Parameters

rhs The other version.

operator> `bool operator>(const ccVersion& rhs) const;`

Returns true if this version is greater than the *rhs*, false otherwise. Use this operator only after verifying that **ccVersion::canCompare(rhs)** is true.

Parameters

rhs The other version.

operator< `bool operator<(const ccVersion& rhs) const;`

Returns true if this version is less than *rhs*, false otherwise. Use this operator only after verifying that **ccVersion::canCompare(rhs)** is true.

Parameters

rhs The other version.

operator>= `bool operator>=(const ccVersion& rhs) const;`

Returns true if this version is greater than or equal to *rhs*, false otherwise. Use this operator only after verifying that **ccVersion::canCompare(rhs)** is true.

Parameters

rhs The other version.

operator<= `bool operator<=(const ccVersion& rhs) const;`

Returns true if this version is less than or equal to *rhs*, false otherwise. Use this operator only after verifying that **ccVersion::canCompare(rhs)** is true.

Parameters

rhs	The other version.

Public Member Functions

getAsText `ccCvlString getAsText() const;`

Returns the version information represented by this object as a formatted text string.

product `ccCvlString product() const;`

Returns the product name, for example, “CVL”.

version `c_UInt32 version() const;`

Returns the version as a 32-bit hexadecimal number, for example, 0x06000000 for CVL 6.0.0.

details `c_UInt32 details() const;`

Return the version details as a 32-bit hexadecimal number, for example, 0x05030100 for CVL 5.3.1.

major `int major() const;`

Retrieves the major version number, for example the 6 in CVL 6.0.0.

minor `int minor() const;`

Retrieves the minor version number, for example the 4 in CVL 5.4.0.

point `int point() const;`

Retrieves the point version, for example the 1 in CVL 5.3.1.

type `ReleaseType type() const;`

Retrieves the release type. This will always be 0 for released software.

sr `int sr() const;`

Retrieves the service release number, for example the 1 in CVL 5.5.2 SR1. Returns zero if this is not a service release.

cr `int cr() const;`

Retrieves the customer release number, for example the 17 in CVL 5.4 CR17. Returns zero if this is not a customer release.

pr `int pr() const;`

Retrieves the patch release number, for example the 6 in CVL 5.4 PR6. Returns zero if this is not a patch release.

build `int build() const;`

Retrieves the build number, for example the 38 in CVL 6.0.0 build 38.

canCompare `bool canCompare(const ccVersion& rhs) const;`

Returns true if this version can be compared to *rhs*. Returns false if the versions cannot be compared, for example for two different products.

Parameters

rhs	The other version.

Macros

cmCvlVersion `cmCvlVersion 0xAABBCCDD`

The CVL version is a 32-bit hexadecimal number defined in *ch_cvl/verdefs.h*. The bytes of the version number specify the following information:

Digits	Meaning
AA	Major version number (for example, the 6 in 6.0.0)
BB	Minor version number (for example, the 4 in 5.4.0)
CC	Point release number (for example, the 1 in 5.3.1)
DD	Release type (always 00 for released software)

cmCvlVersionDetails

```
cmCvlVersionDetails     0xAABBCCDD
```

The CVL version details number is a 32-bit hexadecimal number defined in *ch_cvl/verdefs.h*. The bytes of the version details number specify the following information:

Digits	Meaning
AA	Service release number, for example the 1 in CVL 5.5.2 SR1.
BB	Customer release number, for example the 17 in CVL 5.4 CR17.
CC	Patch release number, for example the 6 in CVL 5.4 PR6.
DD	Build number, for example the 38 in CVL 6.0.0 build 38.

cmCvlVersionMajor

```
cmCvlVersionMajor(cmCvlVersion);
```

Extracts the major version number from the supplied version, for example the 6 in CVL 6.0.0.

cmCvlVersionMinor

```
cmCvlVersionMinor(cmCvlVersion);
```

Extracts the minor version number from the supplied version, for example the 4 in CVL 5.4.0.

cmCvlVersionPoint

```
cmCvlVersionPoint(cmCvlVersion);
```

Extracts the point release number from the supplied version, for example the 1 in CVL 5.3.1.

cmCvlVersionType

```
cmCvlVersionType(cmCvlVersion);
```

Extracts the release type from the supplied version. Always 0 for released software.

ccVersion

cmCvlVersionSr

```
cmCvlVersionSr(cmCvlVersionDetails);
```

Extracts the service release from the supplied CVL version details number, for example the 1 in CVL 5.5.2 SR1.

cmCvlVersionCr

```
cmCvlVersionCr(cmCvlVersionDetails);
```

Extracts the customer release from the supplied CVL version details number, for example the 17 in CVL 5.4 CR17.

cmCvlVersionPr

```
cmCvlVersionPr(cmCvlVersionDetails);
```

Extracts the patch release from the supplied CVL version details number, for example the 6 in CVL 5.4 PR6.

cmCvlVersionBuild

```
cmCvlVersionBuild(cmCvlVersionDetails);
```

Extracts the build number from the supplied CVL version details number, for example the 38 in CVL 6.0.0 build 38.

Note To get version information at run time, call **cfGetRunTimeCvlVersion()** (see **cfGetRunTimeCvlVersion()** on page 3689). To get version information at compile time, call **cfGetRunTimeCvlVersion()** (see **cfGetCompileTimeCvlVersion()** on page 3685).

ccVideoFormat

```
#include <ch_cvl/vidfmt.h>

class ccVideoFormat;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

This is an abstract base class that the acquisition software uses to describe operations on video formats. The acquisition software defines a video format as the pairing of a particular camera type with a particular image size.

The acquisition software creates a single object of class **ccStdVideoFormat** (see page 3009) for each camera type and image size. You use these video format objects to create an acquisition FIFO that is appropriate for the camera and frame grabber that your application uses.

To get a video format, you use **ccStdVideoFormat::getFormat()** described on page 3012.

Notes

There is no need for you to use this class directly in your code. Use the **ccStdVideoFormat** class for all of your format programming.

Constructors/Destructors

ccVideoFormat

```
virtual ~ccVideoFormat();
```

Destructor.

The acquisition software creates one object for each supported video format. You do not create video formats yourself.

Operators

operator==

```
bool operator==(const ccVideoFormat& that);
```

Returns true if this video format is the same as *that* video format.

Parameters

that	The other video format.

operator!=

```
bool operator!=(const ccVideoFormat& that);
```

Returns true if this video format and *that* video format are different.

Parameters

that The other video format.

Notes

Since there is only one instance of each video format, testing video formats for equality is very quick and very simple.

Public Member Functions

width

```
c_Int32 width() const;
```

Returns the width of the video format in pixels.

height

```
c_Int32 height() const;
```

Returns the height of the video format in pixels.

Notes

Due to the limitations of some frame grabbers, it may be necessary to acquire images that are slightly larger or slightly smaller than the image that comes from the camera. The format name always reflects the size of the image coming from the camera. For images returned by **ccAcqFifo**, **ccPelBuffer<P>::width()** and **ccPelBuffer<P>::height()** reflect the actual size of the acquired image.

If you are using a GigE Vision camera with a generic GigE video format, both **width()** and **height()** return 1. To determine the actual size of the acquired image, get the region of interest immediately after creating the acquisition FIFO like this:

```
ccPelRect imageSize(fifo->properties().actualRoi());
```

depth

```
c_Int32 depth() const;
```

Return the depth of the video format in bits per pixel.

newAcqFifo

```
ccAcqFifo* newAcqFifo(ccFrameGrabber& fg);
```

Creates and returns a new acquisition FIFO suitable for acquiring images of this format from the frame grabber *fg*.

Parameters

fg The frame grabber to use to acquire images.

Throws

ccVideoFormat::NotSupported
The specified frame grabber does not support this video format.

ccAcqFifo::CouldNotCreate
An error occurred during creation of the acquisition FIFO.

name

```
const TCHAR* name() const;
```

Returns the name of this video format as a printable string. For example, "Sony XC75 640x480".

cameraManufacturer

```
const TCHAR* cameraManufacturer() const;
```

Returns the name of the manufacturer for the camera that this video format describes. For example, “Sony”.

Notes

NULL is returned if this video format is not specific to a particular camera.

This string is provided for information purposes only, and the string returned by a particular video format should not be relied upon to remain constant from release to release.

cameraModel

```
const TCHAR* cameraModel() const;
```

Returns the name of the model for the camera that this video format describes. For example: “XC-55”.

Notes

NULL is returned if this video format is not specific to a particular camera.

This string is provided for information purposes only, and the string returned by a particular video format should not be relied upon to remain constant from release to release.

videoFormatResolution

```
const TCHAR* videoFormatResolution() const;
```

Returns the text string describing the default resolution of the pel buffer returned by this video format. For example: “640x480”.

Notes

NULL may be returned for some formats.

This string is provided for information purposes only, and the string returned by a particular video format should not be relied upon to remain constant from release to release.

videoFormatDriveType

```
const TCHAR* videoFormatDriveType() const;
```

Returns the text string describing the drive type (internal or external) to be used with this camera (if any). For example: “IntDrv”.

Notes

NULL may be returned when syncModel property is used to control the drive type of this video format, or the sync mode is not applicable.

This string is provided for information purposes only, and the string returned by a particular video format should not be relied upon to remain constant from release to release.

videoFormatOptions

```
const TCHAR* videoFormatOptions() const;
```

Returns the text string defining the operation mode of the camera. Some cameras have several operation modes, so this text string defines the mode that should be used with this video format. For example, “rapid-reset, shutter-sw-EDONPISHAII”.

Notes

NULL may be returned when no special option of the camera is used.

This string is provided for information purposes only, and the string returned by a particular video format should not be relied upon to remain constant from release to release.

formatFromCCF `bool formatFromCCF() const;`

Returns true when the video format was created from a CCF file, or false when the video format is built into CVL.

isSupportedForLegacy

```
bool isSupportedForLegacy(ccFrameGrabber&) const;
```

Returns true when this format is only provided for backwards compatibility reasons on the given frame grabber, but is implemented internally by means of an alias to a CCF video format.

Example

This would return true for the **cfXc55_640x480()** format when used on the MVS-8120/CVM1. The frame grabber actually uses the CCF version of this format in order to perform acquisitions.

Static Functions

fullList

```
static const cmStd vector<const ccVideoFormat*>&
  fullList();
```

Returns a vector of pointers to all supported **ccVideoFormat** objects.

Example

To print out a list of the names of all video formats you might use the following code:

```
#include <ch_cvl/vidfmt.h>
#include <ch_cvl/constrea.h>

typedef cmStd vector<const ccVideoFormat*> Vfv;

void printVideoFormats()
{
  Vfv videoFormats;
  videoFormats = ccVideoFormat::fullList();

  cogOut << cmT("The full list contains ") <<
    videoFormats.size() << cmT(" elements.") << cmStd endl;

  for (Vfv::const_iterator i = videoFormats.begin();
       i != videoFormats.end(); i++)
    cogOut << (*i)->name() << cmStd endl;
}
```

filterList

```
static cmStd vector<const ccVideoFormat*>
  filterList(const cmStd vector<const ccVideoFormat*>& vfs,
  ccFrameGrabber& fg);
```

Given a vector of video formats *vfs*, returns a list of those video formats supported by the frame grabber *fg*.

Parameters

vfs	A vector of video formats.
fg	A frame grabber.

Example

To return a list of all the video formats that support the frame grabber *fg*, you would use the following code:

```
cmStd vector<const ccVideoFormat*>myVidFmts =
  ccVideoFormat::filterList(ccVideoFormat::fullList(), fg);
```

Note

The **ccVideoFormat::filterList()** function returns the Sony XC-55 format as being supported on the MVS-8100. This format is supported only on the MVS-8100M but not the MVS-8100.

ccWaferPreAlign

```
#include <ch_cvl/prealign.h>

class ccWaferPreAlign: public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	Yes
Archiveable	Complex

A class that contains a Wafer Pre-Align tool. A **ccWaferPreAlign** contains the expected diameter and type of the wafer. A **ccWaferPreAlign** must be trained before it can find a wafer.

Constructors/Destructors

ccWaferPreAlign

```
ccWaferPreAlign(double diameter = 200,
  ccWaferPreAlignDefs::FeatureType feature =
  ccWaferPreAlignDefs::eNone,
  ccWaferPreAlignDefs::OperatingMode opMode =
  ccWaferPreAlignDefs::eAutoDetect);

virtual ~ccWaferPreAlign();

ccWaferPreAlign(const ccWaferPreAlign& src);
```

- ```
  ccWaferPreAlign(double diameter = 200,
    ccWaferPreAlignDefs::FeatureType feature =
    ccWaferPreAlignDefs::eNone,
    ccWaferPreAlignDefs::OperatingMode opMode =
    ccWaferPreAlignDefs::eAutoDetect);
  ```

  Constructs a tool in an untrained state.

  Parameters

  diameter — The expected diameter of the wafer in client units. The default value of 200 assumes that client coordinates are calibrated in millimeters.

  feature — The expected feature type of the wafer. *feature* must be one of the following values:
  ```

ccWaferPreAlignDefs::eFlat
ccWaferPreAlignDefs::eNotch
ccWaferPreAlignDefs::eNone

opMode The tool operating mode. *opMode* must be one of the following values:

ccWaferPreAlignDefs::eNotchMax
ccWaferPreAlignDefs::eStandard
ccWaferPreAlignDefs::eAutoDetect

- `virtual ~ccWaferPreAlign();`

  The **ccWaferPreAlign** destructor is virtual, indicating that you may create derived classes from this one.

- `ccWaferPreAlign(const ccWaferPreAlign& src);`

  Copy constructor.

  Parameters
  src The source **ccWaferPreAlign**.

Operators

operator= `ccWaferPreAlign& operator=(const ccWaferPreAlign& src);`

Assignment operator.

Parameters
src The source **ccWaferPreAlign**.

Public Member Functions

diameter

```
double diameter() const;
void diameter(double diameter);
```

- `double diameter() const;`

  Returns the expected diameter of the wafer in client units.

- `void diameter(double diameter);`

  Sets the expected diameter of the wafer in client units. If you call this function, the tool is placed in an untrained state.

  Parameters

  diameter The expected wafer diameter in client units.

  Notes

  This parameter can be set by the tool during learning.

clientFromImageXform

```
cc2Xform clientFromImageXform() const;
```

Returns a **cc2Xform** that describes the transformation between image and client units. This transformation is set when you call train and is typically used to account for any aspect ratio changes between the image coordinates and client coordinates.

featureType

```
ccWaferPreAlignDefs::FeatureType featureType() const;

void featureType(
  ccWaferPreAlignDefs::FeatureType feature);
```

- `ccWaferPreAlignDefs::FeatureType featureType() const;`

  Returns the type of wafer that this **ccWaferPreAlign** is configured to find. This function returns one of the following values:

  ccWaferPreAlignDefs::eFlat
  ccWaferPreAlignDefs::eNotch
  ccWaferPreAlignDefs::eNone

- `void featureType(`
  `ccWaferPreAlignDefs::FeatureType feature);`

  Sets the type of wafer that this **ccWaferPreAlign** is configured to find. You can change the feature type without needing to retrain the **ccWaferPreAlign**.

  Parameters

  feature The type of wafer to find. *feature* must be one of the following values:

  ccWaferPreAlignDefs::eFlat
  ccWaferPreAlignDefs::eNotch
  ccWaferPreAlignDefs::eNone

Notes

This parameter can be set by the tool during learning.

operatingMode

```
ccWaferPreAlignDefs::OperatingMode operatingMode() const;
```

Returns the operating mode for this **ccWaferPreAlign**. This function returns one of the following values:

ccWaferPreAlignDefs::eNotchMax
ccWaferPreAlignDefs::eStandard
ccWaferPreAlignDefs::eAutoDetect

train

```
void train(const cc2Xform &clientFromImageXform);
```

Train a synthetic wafer model. The model is trained using the diameter specified during construction or through a call to **diameter**.

Parameters

clientFromImageXform

A **cc2Xform** describing the transformation between image coordinates and client coordinates.

Throws

ccSecurityViolation

The security information on the Cognex hardware device does not correspond to the operating mode specified for this **ccWaferPreAlign** *object.*

Notes

In most cases, *clientFromImageXform* should be the same **cc2Xform** produced when you calibrated your system. The aspect ratio of *clientFromImageXform* must be identical to the aspect ratio of the client coordinate system associated with images used for learning or searching.

isTrained

```
bool isTrained() const;
```

Returns true if this **ccWaferPreAlign** is currently trained, false otherwise.

learn

```
void learn(const ccPelBuffer_const<c_UInt8>& image,
  const ccWaferPreAlignRunParams& runParams,
  c_UInt32 learnFlags, ccWaferPreAlignResult& result);
```

Analyzes the supplied image of a wafer and determines the wafer's diameter, feature type, or both (as you specify)

The tool sets this **ccWaferPreAlign** object's diameter and feature type. The tool then locates the wafer in the image and places its size, location, and orientation in the supplied **ccWaferPreAlignResult** object.

You must call **train()** before you call this function, and the **cc2Xform** that you supply to **train()** must have the same aspect ratio as the client coordinate system of the image that you supply to this function. A simple way of making sure the aspect ratios are the same is to supply use the same image for both **train()** and **learn()**, as shown in the following example:

```
ccPelBuffer_const<c_UInt8> image;
ccWaferPreAlignRunParams params;
ccWaferPreAlignResult results;

...

void train(image.clientFromImageXform());
void learn(image, params, ccWaferPreAlignDefs::eAll, results);
```

Parameters

image An image of the wafer. The wafer should be backlit and it should occupy approximately 90% of *image*'s area.

runParams A **ccWaferPreAlignRunParams** that specifies the parameters used to align the wafer.

learnFlags Which wafer parameters to learn. *learnFlags* must be formed by ORing together one or more of the following values:

ccWaferPreAlignDefs::eDiameter
ccWaferPreAlignDefs::eFeature
ccWaferPreAlignDefs::eAll

result A **ccWaferPreAlignResult** into which the results of the pre-alignment are placed.

Throws

ccWaferPreAlignDefs::BadParams
learnFlags is not formed by ORing together one or more of *ccWaferPreAlignDefs::eDiameter*, *ccWaferPreAlignDefs::eFeature*, or *ccWaferPreAlignDefs::eAll* or *runParams* does not contain valid parameters.

ccWaferPreAlignDefs::BadImage
image is unbound.

ccWaferPreAlignDefs::BadXform
: The aspect ratio specified by *image*'s client coordinates is not the same as the aspect ratio that you supplied to the **train()** function.

ccSecurityViolation
: The security information on the Cognex hardware device does not correspond to the operating mode specified for this **ccWaferPreAlign** *object.*

run

```
void run(const ccPelBuffer_const<c_UInt8>& image,
  const ccWaferPreAlignRunParams& runParams,
  ccWaferPreAlignResult& result) const;
```

Attempts to locate a wafer of the diameter and type specified by this **ccWaferPreAlign** using the supplied **ccWaferPreAlignRunParams**. The results of the alignment operation are placed in the supplied **ccWaferPreAlignResult**.

This function first locates the center of the wafer. Then, if the wafer type is flat or notched, the tool determines the location of the alignment feature on the wafer.

You must call **train()** before you call this function, and the **cc2Xform** that you supply to **train()** must have the same aspect ratio as the client coordinate system of the image that you supply to this function. A simple way of making sure the aspect ratios are the same is to use the same image for both **train()** and **learn()**, as shown in the following example:

```
ccPelBuffer_const<c_UInt8> image;
ccWaferPreAlignRunParams params;
ccWaferPreAlignResult results;
ccWaferPreAlign wpa;

...

wpa.train(image.clientFromImageXform());
wpa.learn(image, params, ccWaferPreAlignDefs::eAll, results);
wpa.run(image, params, results);
```

Parameters

image
: An image of the wafer. The wafer should be backlit and it should occupy approximately 90% of *image*'s area.

runParams
: A **ccWaferPreAlignRunParams** which specifies the parameters for the pre-alignment.

result
: A **ccWaferPreAlignResult** into which the results of the pre-alignment are placed.

Throws

ccSecurityViolation
The security information on the Cognex hardware device does not correspond to the operating mode specified for this **ccWaferPreAlign** *object.*

ccWaferPreAlignDefs::BadParams
runParams contains an invalid combination of run-time parameters.

ccWaferPreAlignDefs::BadImage
image is unbound.

ccWaferPreAlignDefs::BadXform
The aspect ratio specified by *image*'s client coordinates is not the same as the aspect ratio that you supplied to the **train()** function.

update

```
virtual bool update(c_UInt32 currDiameter,
  double currAspectRatio,
  ccWaferPreAlignDefs::FeatureType currFeature,
  c_UInt32 bestDiameter, double bestAspectRatio,
  ccWaferPreAlignDefs::FeatureType bestFeature,
  const ccWaferPreAlignResult &currResult,
  const ccWaferPreAlignResult &bestResult);
```

This function is called during the learning process by the Wafer Pre-Align tool. You can override this function in a class derived from **ccWaferPreAlign** to monitor the progress of the learning operation. You can terminate the learning process by having your override return false. Returning true continues the learning process.

Parameters

currDiameter The most recently evaluated wafer diameter in client units.

currAspectRatio The most recently evaluated aspect ratio.

currFeature The most recently evaluated feature type. *currFeature* is one of the following values:

ccWaferPreAlignDefs::eFlat
ccWaferPreAlignDefs::eNotch
ccWaferPreAlignDefs::eNone

bestDiameter The wafer diameter that has produced the best result so far.

bestAspectRatio The aspect ratio that has produced the best result so far.

bestFeature The feature type that has produced the best result so far. *bestFeature* is one of the following values:

ccWaferPreAlignDefs::eFlat
ccWaferPreAlignDefs::eNotch
ccWaferPreAlignDefs::eNone

Notes

You should not call this function. You use this function by overriding it in a derived class.

ccWaferPreAlignDefs

```
#include <ch_cvl/prealign.h>

class ccWaferPreAlignDefs;
```

A name space that holds enumerations and constants used by the Wafer Pre-Align tool.

Enumerations

FeatureType

```
enum FeatureType;
```

This enumeration defines types of wafers that can be located by the Wafer Pre-Align tool.

Value	Meaning
eFlat	Round wafer with an alignment flat
eNotch	Round wafer with an alignment notch
eNone	Round wafer with no alignment feature

LearnFlags

```
enum LearnFlags;
```

This enumeration defines which wafer parameters will be learned by automatic learning.

Value	Meaning
eDiameter	Learn the diameter of the wafer.
eFeature	Learn the feature type (alignment flat, alignment notch, or no alignment feature)
eAll	Learn both the diameter and the feature type.

OperatingMode `enum OperatingMode;`

This enumeration defines the operating modes supported by the Wafer Pre-Align Tool. Keep in mind that your Cognex hardware must have the security bit enabled that corresponds to the operating mode you request.

Value	Meaning
eNotchMax	Use the NotchMax operating mode. This is a general-purpose operating mode for front-lit wafers.
eStandard	Use the Standard operating mode. This mode requires that the wafer be backlit and that the input image contain a silhouette of the wafer.
eAutoDetect	If the Cognex hardware has NotchMax enabled, the tool uses the NotchMax operating mode regardless of whether Standard mode is enabled. If only Standard mode is enabled, the tool uses Standard operating mode.

ccWaferPreAlignResult

```
#include <ch_cvl/prealign.h>

class ccWaferPreAlignResult : public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that contains the results produced by the Wafer Pre-Align tool.

Constructors/Destructors

ccWaferPreAlignResult

```
ccWaferPreAlignResult();
```

Constructs a **ccWaferPreAlignResult** with the following default values:

Parameter	Value
waferFoundFlag	False
featureFoundFlag	False
featureType	*eNone*
center	0.0, 0.0
angle	0.0
scale	1.0
score	0.0
time	0.0
imgFromClient	Identity transform
featureLocation	0.0, 0.0
notchDepth	0.0
notchWidth	0.0

Parameter	Value
flatLength	0.0
flatRadius	0.0

Public Member Functions

isWaferFound

```
bool isWaferFound() const;
```

Returns true if a wafer was found, false otherwise. In NotchMax mode, this value is true if the score is 1.0, false if the score is 0.0. In Standard mode, this value is true if the score equals or exceeds the accept threshold you specified.

isFeatureFound

```
bool isFeatureFound() const;
```

Returns true if both the wafer and the feature specified in the **ccWaferPreAlign** that was run are found, false otherwise. If the **ccWaferPreAlign** was configured to find a round wafer, this function returns true if the wafer is found.

featureType

```
ccWaferPreAlignDefs::FeatureType featureType() const;
```

Returns the type of feature specified by the **ccWaferPreAlign** that was run.

center

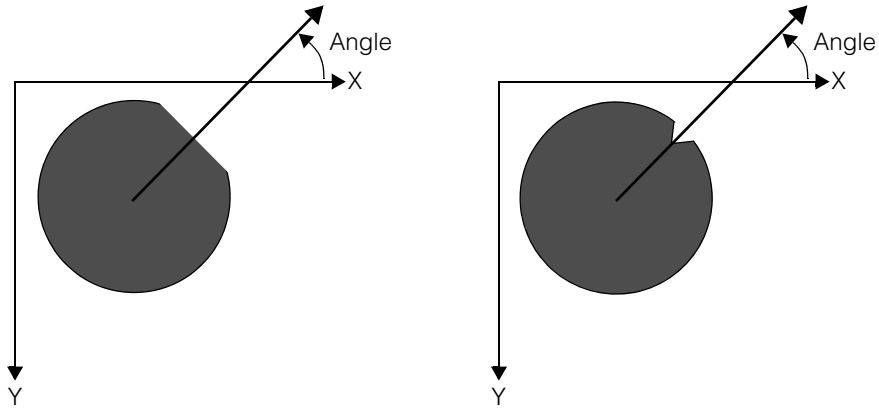
```
cc2Vect center() const;
```

Returns the location of the center of the found wafer. The wafer center is reported in the client coordinate system, offset by the origin specified in the **ccWaferPreAlignRunParams** used for this search.

angle

```
ccRadian angle() const;
```

Returns the angle of the notch or the midpoint of the flat area. The angle is reported as the angle from the client coordinate system x-axis to a line drawn from the wafer center through the center of the notch or flat, as shown in the following diagram:



If this **ccWaferPreAlign** is configured to locate round wafers, then this function returns 0.0.

Throws

ccWaferPreAlignDefs::NotFound
The specified feature type was not found.

scale

```
double scale() const;
```

Returns the scale of the found wafer as a multiple of the size of the trained wafer.

score

```
double score() const;
```

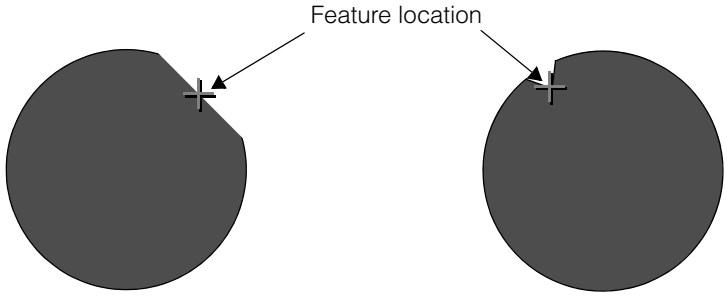
In Standard mode, this function returns the result score, a number between 0.0 and 1.0, where 1.0 represents a perfect match and 0.0 represents a total failure. In NotchMax mode, the score is either 0.0 (failure) or 1.0 (success).

Notes

In Standard mode, if the score is determined to be 0.0 while locating the wafer, the feature locating step is not performed. The Standard mode score considers only the perimeter of the wafer, not the feature. A wafer can have a high score (a good perimeter) even if no feature is found.

featureLocation `cc2Vect featureLocation() const;`

Returns the location of the feature with respect to the origin of the client coordinate system. For notched wafers, the feature location is the deepest point in the notch. For flat wafers, the feature location is the center of the flat. The following diagram shows how the feature location is reported:



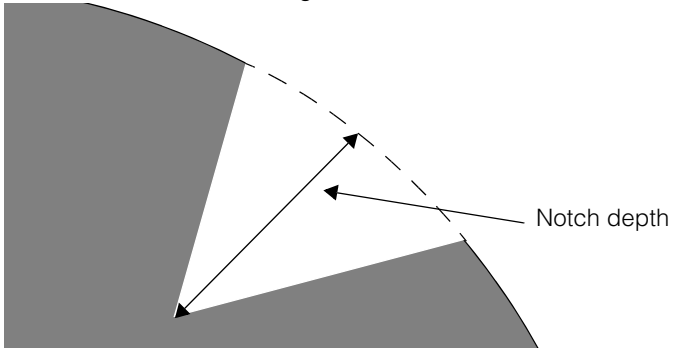
Throws

ccWaferPreAlignDefs::WrongFeatureType
This **ccWaferPreAlign** is configured to locate round wafers

ccWaferPreAlignDefs::NotFound
The specified feature type was not found.

notchDepth `double notchDepth() const;`

Returns the distance from the nominal perimeter of the wafer to the deepest point on the found notch measured along the wafer radius, as shown in the following figure:



The distance is returned in client units. This is equal to the difference between the wafer radius and the distance from the wafer center to the deepest point on the found notch.

Throws

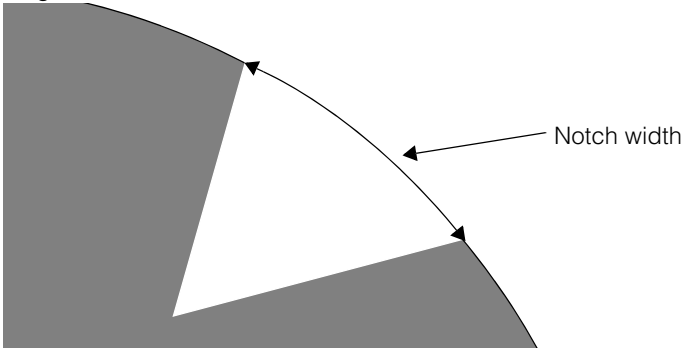
ccWaferPreAlignDefs::WrongFeatureType
This **ccWaferPreAlign** is not configured to find notched wafers.

ccWaferPreAlignDefs::NotFound
The specified feature type was not found.

notchWidth

```
double notchWidth() const;
```

Returns the distance from one corner of the found notch to the other measured along the wafer's circumference in the client coordinate system, as shown in the following diagram:



Throws

ccWaferPreAlignDefs::WrongFeatureType
This **ccWaferPreAlign** is not configured to find notched wafers.

ccWaferPreAlignDefs::NotFound
The specified feature type was not found.

flatLength

```
double flatLength() const;
```

Returns the length of the found flat area in the client units.

Throws

ccWaferPreAlignDefs::WrongFeatureType
This **ccWaferPreAlign** is not configured to find flat wafers.

ccWaferPreAlignDefs::NotFound
The specified feature type was not found.

flatRadius

```
double flatRadius() const;
```

Returns the shortest distance between the wafer center and the wafer's flat section in the client units.

Throws

ccWaferPreAlignDefs::WrongFeatureType
This **ccWaferPreAlign** is not configured to find flat wafers.

ccWaferPreAlignDefs::NotFound
The specified feature type was not found.

time

```
double time() const;
```

Returns the time in seconds required to locate the wafer and the feature.

imageFromClientXform

```
const cc2Xform& imageFromClientXform () const;
```

Returns the **cc2Xform** that converts client coordinate system locations to image coordinate system locations.

updateWaferResults

```
void updateWaferResults(bool waferFound,
  const cc2Vect &center, const ccRadian &angle,
  double scale, double score, const cc2Xform& xform);
```

Sets the following values in this **ccWaferPreAlignResult**:

- Whether a wafer was found
- The wafer location
- The wafer angle
- The wafer scale
- The wafer score
- The image-to-client transformation.

This function is intended to let you manually set the values in a **ccWaferPreAlignResult** for the purposes of testing your application.

Parameters

waferFound If true, the wafer was found, if false it was not.

scale The scale of the found wafer as a multiple of the size of the trained wafer.

center	The location of the center of the wafer in client coordinates.
angle	The angle from the client coordinate system x-axis to a line drawn from the wafer center through the center of the notch or flat.
score	The score received by the wafer. *score* must be in the range of 0.0 through 1.0.
xform	A **cc2Xform** that converts client coordinate system locations to image coordinate system locations.

updateFeatureResults

```
void updateFeatureResults(bool featureFound,
  ccWaferPreAlignDefs::FeatureType feature,
  const cc2Vect &location, const ccRadian &angle,
  double notchDepth, double notchWidth, double flatLength,
  double flatRadius, double time);
```

Sets the following values in this **ccWaferPreAlignResult**:

- Whether the requested feature type was found
- The feature type
- The feature location
- The notch depth and width
- The flat length and radius
- The orientation of the wafer
- The amount of time required to locate the wafer and feature.

This function is intended to let you manually set the values in a **ccWaferPreAlignResult** for the purposes of testing your application.

Parameters

featureFound	If true, the specified feature was found, if false it was not.
feature	The type of feature. *feature* must be one of the following values: *ccWaferPreAlignDefs::eFlat* *ccWaferPreAlignDefs::eNotch* *ccWaferPreAlignDefs::eNone*
location	The location of the feature in the client coordinate system offset.
angle	The angle from the client coordinate system x-axis to a line drawn from the wafer center through the center of the notch or flat.

notchDepth	The depth of the notch in client units, measured from the nominal wafer perimeter to the deepest point of the notch along the radius of the wafer.
notchWidth	The width of the notch in client units, measured along the nominal wafer perimeter.
flatLength	The length of the flat in client units.
flatRadius	The minimum distance from the flat to the center of the wafer in client units.
time	The number of seconds required to find the wafer and feature.

ccWaferPreAlignRunParams

```
#include <ch_cvl/prealign.h>

class ccWaferPreAlignRunParams : public ccPersistent;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

A class that contains the run-time parameters used by the Wafer Pre-Align tool.

Many of the run-time parameters defined in this class apply only to Standard mode; in NotchMax mode, those parameters are ignored. The following table lists which parameters are used in the two modes.

Parameter	Standard Mode	NotchMax Mode
origin	Yes	Yes
accept	Yes	No
scale	Yes	Yes
scaleRange	Yes	Yes
maxEccentricity	Yes	No
minNotchDepth	Yes	No
maxNotchDepth	Yes	No
minNotchWidth	Yes	No
maxNotchWidth	Yes	No
minFlatRadius	Yes	No
maxFlatRadius	Yes	No
minFlatLength	Yes	No
maxFlatLength	Yes	No

Constructors/Destructors

ccWaferPreAlignRunParams

```
ccWaferPreAlignRunParams();
```

Constructs **ccWaferPreAlignRunParams** with the following default values:

Parameter	Default value
origin	0.0, 0,0
accept	0.3
scale	1.0
scaleRange	0.0
maxEccentricity	0.0
minNotchDepth	0.0
maxNotchDepth	0.0
minNotchWidth	0.0
maxNotchWidth	0.0
minFlatRadius	0.0
maxFlatRadius	0.0
minFlatLength	0.0
maxFlatLength	0.0

Public Member Functions

origin

```
cc2Vect origin() const;
void origin(const cc2Vect &origin);
```

- `cc2Vect origin() const;`

  Returns the origin in client coordinates. The wafer position is reported relative to the origin that you specify.

- `void origin(const cc2Vect &origin);`

  Sets the origin in client coordinates. The wafer position is reported relative to the origin that you specify.

  Parameters

  origin The origin in client coordinates

accept

```
double accept() const;
void accept(double accept);
```

- `double accept() const;`

  Returns the accept threshold. Only wafers that receive a score greater than or equal to the accept threshold are returned.

- `void accept(double accept);`

  Sets the accept threshold. Only wafers that receive a score greater than or equal to the accept threshold are returned.

  Parameters

  accept The accept threshold. *accept* must be in the range 0.0 through 1.0.

  Throws

  ccWaferPreAlignDefs::BadParams

  accept is not in the range 0.0 through 1.0.

scale

```
double scale() const;
void scale(double scale);
```

- `double scale() const;`

  Returns the expected scale of the wafer.

- `void scale(double scale);`

  Sets the expected scale of the wafer.

  You specify a tolerance range for the scale using **scaleRange**.

Parameters

scale The expected scale of the wafer, expressed as the scale of the found wafer with respect to the trained wafer. A value of 1.25 specifies that the found wafer will be 25% larger than the trained wafer.

Throws

ccWaferPreAlignDefs::BadParams
scale is less than or equal to 0.0.

---

scaleRange

```
double scaleRange() const;
void scaleRange(double range);
```

- `double scaleRange() const;`

  Returns one half of the total scale tolerance.

- `void scaleRange(double range);`

  Sets the scale tolerance range. You specify the range as the permitted positive or negative scale variation from the expected scale (specified with **scale()**).

  To specify a scale range of 1.0 to 1.5, you would specify a value of 1.25 to **scale()** and a value of 0.25 to **scaleRange()**.

  Parameters

  range The maximum positive or negative scale difference from the expected scale of the wafer.

  Throws

  ccWaferPreAlignDefs::BadParams
  range is less than 0.0.

---

maxEccentricity

```
double maxEccentricity() const;
void maxEccentricity(double maxEccentricity);
```

- `double maxEccentricity() const;`

  Returns the maximum expected deviation of the shape of the wafer from a circle.

- `void maxEccentricity(double maxEccentricity);`

  Sets the maximum expected deviation of the shape of the wafer from a circle.

  When *maxEccentricity* is set to 0.0, the wafer is expected to appear as a perfect circle. As *maxEccentricity* is increased, the wafer is assumed to become progressively more elliptical.

  The Wafer Pre-Align tool uses this value to help determine the wafer boundary. Larger values of *maxEccentricity* may make the tool more sensitive to the effects of video noise.

  maxEccentricity is computed using the following formula:

$$maxEccentricity = \sqrt{1 - \frac{min^2}{max^2}}$$

  where *min* is the length of the minor principal axis and *max* is the length of the major principal axis.

  Parameters

  maxEccentricity The maximum expected eccentricity, as computed above.

  Throws

  ccWaferPreAlignDefs::BadParams
  maxEccentricity is less than 0.0 or greater than 0.25.

minNotchDepth `double minNotchDepth() const;`

Returns the minimum expected depth of the notch feature in client units.

maxNotchDepth `double maxNotchDepth() const;`

Returns the maximum expected depth of the notch feature in client units.

notchDepthRange

`void notchDepthRange(double minimum, double maximum);`

Sets the minimum and maximum expected depth of the notch feature in client units. The notch depth is measured from the point within the notch that is closest to the wafer center to the nominal perimeter of the wafer. This is equal to the difference between the wafer radius and the distance from the wafer center to the deepest point on the notch.

Parameters

minimum The minimum expected notch depth.

maximum The maximum expected notch depth.

Throws

ccWaferPreAlignDefs::BadParams

minimum or *maximum* is less than 0.0 or *minimum* is greater than *maximum*.

Notes

These parameters are only used for wafer notch location and are not considered for the location of other features.

minNotchWidth

```
double minNotchWidth() const;
```

Returns the minimum expected width of the notch feature in client units.

maxNotchWidth

```
double maxNotchWidth() const;
```

Returns the maximum expected width of the notch feature in client units.

notchWidthRange

```
void notchWidthRange(double minimum, double maximum);
```

Sets the minimum and maximum expected width of the notch feature in client units. The notch width is defined as the distance between the two corners of the notch measured along the wafer's circumference.

Parameters

minimum The minimum expected notch width.

maximum The maximum expected notch width.

Throws

ccWaferPreAlignDefs::BadParams

minimum or *maximum* is less than 0.0 or *minimum* is greater than *maximum*.

Notes

These parameters are only used for wafer notch location and are not considered for the location of other features.

minFlatLength

```
double minFlatLength() const;
```

Returns the minimum expected length of the flat feature in client units.

maxFlatLength

```
double maxFlatLength() const;
```

Returns the maximum expected length of the flat feature in client units.

flatLengthRange

```
void flatLengthRange(double minimum, double maximum);
```

Sets the minimum and maximum expected length of the flat feature in client units.

Parameters

minimum The minimum expected flat length.

maximum The maximum expected flat length.

Throws

ccWaferPreAlignDefs::BadParams

minimum or *maximum* is less than 0.0 or *minimum* is greater than *maximum*.

Notes

These parameters are only used for wafer flat location and are not considered for the location of other features.

minFlatRadius

```
double minFlatRadius() const;
```

Returns the minimum expected distance from the wafer center to the center of the flat in client units.

maxFlatRadius

```
double maxFlatRadius() const;
```

Returns the maximum expected distance from the wafer center to the center of the flat in client units.

flatRadiusRange

```
void flatRadiusRange(double minimum, double maximum);
```

Sets the minimum and maximum expected distance from the wafer center to the center of the flat in client units.

Parameters

minimum The minimum expected distance.

maximum The maximum expected distance.

Throws

ccWaferPreAlignDefs::BadParams

minimum or *maximum* is less than 0.0 or *minimum* is greater than *maximum*.

Notes

These parameters are only used for wafer flat location and are not considered for the location of other features.

ccWin32Display

```
#include <ch_cvl/w32disp.h>

class ccWin32Display : public ccDisplay;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	No

The **ccWin32Display** class provides a display window for images and graphics you can program using the Microsoft Win32 API for execution on a Windows host computer. **ccWin32Display** is intended for Microsoft Windows applications that do not use MFC, or for any application that requires a simple display window. **ccWin32Display** windows do not include a title bar, menu bar, tool bar, and status bar like those built into **ccDisplayConsole** windows. The class does provide scroll bars and methods you can call for zooming, panning, and other features.

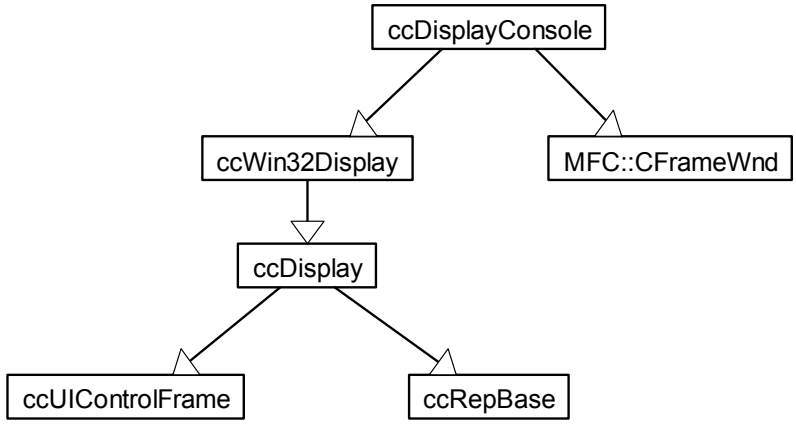


Figure 1. ccWin32Display class inheritance hierarchy

When you build a **ccWin32Display** application, the project's preprocessor settings should define the macro **cmNoMFCDependency** to avoid including MFC header files.

If you are programming under MFC, you may wish to use the **ccDisplayConsole** class which provides a full-featured display and drawing environment and graphical user interface, all through MFC. Tool bars, status bars, and many other GUI features are provided by MFC that may improve your application's user interface.

The **ccWin32Display** class has the following capabilities:

- Displaying images (pel buffers)
- Displaying live images from Cognex frame grabbers
- Adding static graphics to the display
- Adding interactive graphics to the display
- GUI with mouse support

Constructors/Destructors

ccWin32Display

```
ccWin32Display(HWND hWnd = 0);
```

Creates a display object using *hWnd* as the display window. *hWnd* is subclassed so that the display can receive Windows messages such as mouse movements, mouse clicks, and window resizing.

Parameters

hWnd Handle to the window to use for the display.

Notes

hWnd must belong to the current process and must not be subclassed by any other **ccWin32Display** object.

Public Member Functions

window

```
void window(HWND hWnd);

HWND window();
```

- `void window(HWND hWnd);`

Specifies the window where the image and graphics should be displayed. This window is subclassed so that the display can receive Windows messages, such as mouse movements, mouse clicks, and window resizing.

Parameters

hWnd Handle to the window to use for the display.

Notes

hWnd must belong to the current process and must not be subclassed by any other **ccWin32Display** object.

A valid window should be set before attempting to use the display.

- `HWND window();`

Returns the handle of the window that this display appears in.

fontTable

```
void fontTable(const cmStd vector<HFONT>& table);
const cmStd vector<HFONT>& fontTable() const;
```

- `void fontTable(const cmStd vector<HFONT>& table);`

Sets the font table. The **ccUIFormat::fontID()** corresponds to the font table index. If an out-of-range font ID is passed to a drawing routine, the system font is used.

Parameters

table The font table

Notes

You are responsible for ensuring the proper *HFONT* lifetime.

- `const cmStd vector<HFONT>& fontTable() const;`

Retrieves the current font table.

showScrollBar

```
void showScrollBar(bool horiz, bool vert);
```

Specifies whether the scroll bars are visible.

Parameters

horiz If true, makes the horizontal scroll bar visible; if false, makes the horizontal scroll bar invisible.

vert If true, makes the vertical scroll bar visible; if false, makes the vertical scroll bar invisible.

horzScrollbarEnabled

```
bool horzScrollbarEnabled() const;
```

Returns true if the horizontal scroll bar is visible.

vertScrollbarEnabled

```
bool vertScrollbarEnabled() const;
```

Returns true if the vertical scroll bar is visible.

enableOverlay

```
overlayState enableOverlay() const;

void enableOverlay(overlayState val);
```

Notes

These overrides implement the **enableOverlay()** methods, inherited from the **ccDisplay** base class, with behavior specific to **ccWin32Display** objects.

- `overlayState enableOverlay() const;`

  Returns the state of the overlay layer on this **ccWin32Display**.

- `void enableOverlay(overlayState state);`

  Enables or disables the overlay layer on this **ccWin32Display**. For MVS-8500 series frame grabbers, you must explicitly enable the overlay layer by calling this function with *ccDisplay::eOverlayPlane* as the argument.

  Parameters

  state — The state of the host graphics overlay layer. Must be one of:

  ccDisplay::eOverlayPlane
  ccDisplay::eDisableOverlayPlane
  ccDisplay::eNone

  Notes

  Overlay state values cannot be ORed together. *ccDisplay::eNone* should never be passed as an argument to **enableOverlay()**. This value can be returned by **enableOverlay()** when the overlay layer is disabled.

  When the overlay layer is disabled (*ccDisplay::eDisableOverlayPlane*), any sketches and interactive shapes added to the overlay layer disappear from the screen. Before disabling the overlay layer, you must remove any interactive shapes currently on the overlay layer and erase all sketches on all layers.

  If there is not enough video memory to allocate to the overlay layer, system memory is used. Using system memory may slow down overlay performance.

  The overlay layer cannot be enabled unless a valid window handle has been set. See **window(HNWD)** for more information.

  See **ccDisplay::enableOverlay()** for more information.

selectAreaStart

```
virtual c_Int32 selectAreaStart(CoordinateSystem cs,
  const ccPelRect& startLocation, const ccColor& color);
```

Starts an interactive area selection and returns a tag used to specify the end of the selection.

Parameters

cs The coordinate system to use. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation The initial location of the selection area in the specified coordinate system. If *startLocation* is a null rectangle or if it does not overlap the visible portion of the image, the initial location of the selection area is the location of the most recently selected area. If the most recently selected area is not visible or if there was no previously selected area, the starting location is two-thirds of the visible area.

color The color to use to draw the selection rectangle.

Example

For selecting an area, **selectArea()** is easier to use than **selectAreaStart()** and **selectAreaEnd()**. However, you can use **selectAreaStart()** and **selectAreaEnd()** to implement specialized versions of **selectArea()**. The following example shows how to write an override of **selectArea()** that the user can cancel without making a selection. Assume that **ccMyWin32Display** is derived from this class.

```
ccPelRect ccMyWin32Display::selectArea(CoordinateSystem cs,
             const ccPelRect &rect, const TCHAR* prompt,
             const ccColor& c)
{
  c_Int32 tag = ccWin32Display::selectAreaStart(cs,rect,c);
  c_Int32 v = MessageBox(NULL, prompt, cmT("Select Area"),
                         MB_OKCANCEL);

  if(v == IDOK){
    return ccWin32Display::selectAreaEnd(tag);
  } else if(v == IDCANCEL) {
    return ccPelRect();
```

```
  } else
    return ccPelRect();
}
```

Notes

If the specified coordinate system is *eClientCoords*, then the client coordinates of the last displayed image are used. If there is no such image, then display coordinates are silently used and no error is returned.

The box manipulated during the drag operation is always image aligned. Therefore, if the specified coordinate system is *eClientCoords* and the client coordinate system contains skew, this skew is not reflected in the drawn rectangle.

selectAreaEnd

```
virtual ccPelRect selectAreaEnd(c_Int32 tag);
```

Completes the area selection started by **selectAreaStart()** by hiding the rectangle graphics.

Parameters

tag — The tag that identifies the selection to finish. If tag is not a value returned by **selectAreaStart()**, this function has no effect.

selectArea

```
virtual ccPelRect selectArea(CoordinateSystem cs,
  const ccPelRect& startLocation,
  const TCHAR* prompt, const ccColor& color);
```

Waits for the user to select a rectangular area on the display and returns the rectangle in the coordinate system specified. The prompt is displayed in a message box. This function does not return until the user confirms their selection in the message box.

Parameters

cs — The coordinate system to use for reporting the location of the selected rectangle. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation — The initial location of the selection area in the specified coordinate system. If *startLocation* is a null rectangle or if it does not overlap the visible portion of the image, the initial location of the selection area is the location of the most recently selected area. If the most recently selected area is not visible or if there was no previously selected area, the starting location is two-thirds of the visible area.

prompt — The prompt that appears in the message box.

color The color to use to draw the selection rectangle.

Notes

If the specified coordinate system is *eClientCoords*, then the client coordinates of the last displayed image are used. If there is no such image, then display coordinates are silently used and no error is returned.

The box manipulated during the drag operation is always image aligned. Therefore, if the specified coordinate system is *eClientCoords* and the client coordinate system contains skew, this skew is not reflected in the drawn rectangle.

selectPointStart

```
virtual c_Int32 selectPointStart(CoordinateSystem cs,
  const ccDPair& startLocation, const ccColor& color);
```

Starts a point selection. The start location is specified in image coordinates or display coordinates.If start location is not in the visible part of the image, the initial location of the cross will be the last selected point (or the center of the display if there was no last selection or if the last selected point would not be visible)

Parameters

cs The coordinate system to use. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation The initial location of the selection cross in the specified coordinate system. If *startLocation* is not in the visible portion of the image, the initial location of the selection cross is the location of the most recently selected point. If the most recently selected point is not visible or if there was no previously selected point, the starting location is the center of the display.

color The color to use to draw the selection cross.

Notes

If the specified coordinate system is *eClientCoords*, then the client coordinates of the last displayed image are used. If there is no such image, then display coordinates are silently used and no error is returned.

selectPointEnd

```
virtual ccDPair selectPointEnd(c_Int32 tag);
```

Completes a point selection started with **selectPointStart()** by hiding the cross graphics.

Parameters

tag — The tag that identifies the selection to finish. If tag is not a value returned by **selectPointStart()**, this function has no effect.

selectPoint

```
virtual ccDPair selectPoint(CoordinateSystem cs,
  const ccDPair& startLocation, const TCHAR* prompt,
  const ccColor& color);
```

Waits for the user to select a point on the display and returns the point in the coordinate system specified. The prompt is displayed in a message box; this function does not return until the user confirms the selection in the message box. The user may cancel the operation instead of selecting.

Parameters

cs — The coordinate system to use. Must be one of:

ccDisplay::eDisplayCoords
ccDisplay::eImageCoords
ccDisplay::eClientCoords

startLocation — The initial location of the selection cross in the specified coordinate system. If *startLocation* is not in the visible portion of the image, the initial location of the selection cross is the location of the most recently selected point. If the most recently selected point is not visible or if there was no previously selected point, the starting location is the center of the display.

prompt — A prompt that appears in a message box.

color — The color to use to draw the selection cross.

Notes

If the specified coordinate system is *eClientCoords*, then the client coordinates of the last displayed image are used. If there is no such image, then display coordinates are silently used and no error is returned.

displayFormat

```
virtual DisplayFormat displayFormat() const;
```

Returns the display format of this display.

startLiveDisplay

```
bool startLiveDisplay(ccGreyAcqFifoPtrh fifo);
bool startLiveDisplay(ccRGB16AcqFifoPtrh fifo);
bool startLiveDisplay(ccRGB32AcqFifoPtrh fifo);
```

Notes

While live display is in progress, **startLiveDisplay()** owns the FIFO. Your application should not make changes to *fifo* until you stop live display.

- `bool startLiveDisplay(ccGreyAcqFifoPtrh fifo);`

Enables live display of images acquired through the specified acquisition FIFO. Returns true if the operation was successful, false otherwise.

Parameters

fifo The acquisition FIFO to use for the live display.

- `bool startLiveDisplay(ccRGB16AcqFifoPtrh fifo);`

Enables live display of images acquired through the specified acquisition FIFO. Returns true if the operation was successful, false otherwise.

Parameters

fifo The acquisition FIFO to use for the live display.

- `bool startLiveDisplay(ccRGB32AcqFifoPtrh fifo);`

Enables live display of images acquired through the specified acquisition FIFO. Returns true if the operation was successful, false otherwise.

Parameters

fifo The acquisition FIFO to use for the live display.

stopLiveDisplay

```
virtual void stopLiveDisplay();
```

Stops the live display thread started by **startLiveDisplay()**.

isLiveEnabled

```
virtual bool isLiveEnabled() const;
```

Returns true if live display has started and is active, otherwise false.

liveFrameRate `virtual double liveFrameRate() const;`

Returns the live display frame rate in frames per second.

hasImage `virtual bool hasImage() const;`

Returns true if the display has an image.

imageSize `virtual ccIPair imageSize() const;`

Returns the image size as a (width, height) pair if the display has an image. Otherwise it returns (0,0).

imageOffset `virtual ccIPair imageOffset() const;`

Returns the offset of the image as an (x,y) pair if the display has an image. Otherwise if returns (0,0).

toolsPopupMenuEnabled

```
bool toolsPopupMenuEnabled() const;
void toolsPopupMenuEnabled(bool enabled);
```

When enabled, a tools pop-up menu appears when you right-click the mouse in a **ccWin32Display** window.

- `bool toolsPopupMenuEnabled() const;`

  Returns true if the tools pop-up menu is enabled.

- `void toolsPopupMenuEnabled(bool enabled);`

  Enables or disables the tools pop-up menu. The pop-up menu contains standard functionality for zooming, panning, etc.

  Parameters

  enabled True if the pop-up menu is enabled.

getDisplayedImage

```
void getDisplayedImage(ccPelBuffer<c_UInt8>& buf,
 ccUITablet::Layers layer=ccUITablet::eUnspecifiedLayer);

void getDisplayedImage(ccPelBuffer<ccPackedRGB16Pel>& buf,
 ccUITablet::Layers layer=ccUITablet::eUnspecifiedLayer);

void getDisplayedImage(ccPelBuffer<ccPackedRGB32Pel>& buf,
 ccUITablet::Layers layer=ccUITablet::eUnspecifiedLayer);
```

Retrieves the image and graphics displayed on the specified layer. The default is a combined image comprising both the image layer and the overlay layer.

The displayed image is stored to the specified pel buffer. If the desktop color depth does not match the pel buffer type, then the image data are automatically converted to the format of the pel buffer that is to receive the data. To prevent loss of data during conversion, choose the overload that matches the current desktop depth.

These functions always use display coordinates (*eDisplayCoords*). That is, the upper left corner of *buf* is the upper left corner of the display, and not the upper left corner of the currently displayed image.

- ```
  void getDisplayedImage(ccPelBuffer<c_UInt8>& buf,
   ccUITablet::Layers layer=ccUITablet::eUnspecifiedLayer);
  ```

  Places the image currently being displayed on this **ccWin32Display** into the supplied **ccPelBuffer<c_UInt8>**.

  Parameters

  buf — A reference to the 8-bit pel buffer into which the image will be copied. Must be presized to the size of the image you want to save.

  layer — If *ccUITablet::eImageLayer*, then only image pixels are copied. If *ccUITablet::eOverlayLayer*, then only graphics from the overlay layer are copied. If *ccUITablet::eUnspecifiedLayer*, then both image and graphics are copied.

  Notes

  Use this overload if your desktop is set to 8 bits.

  Throws

  OverlayLayerNotEnabled — The overlay layer is not enabled and the specified layer is *ccUITablet::eOverlayLayer*.
  ```

- `void getDisplayedImage(ccPelBuffer<ccPackedRGB16Pel>& buf, ccUITablet::Layers layer=ccUITablet::eUnspecifiedLayer);`

  Places the image currently being displayed on this **ccWin32Display** into the supplied **ccPelBuffer<ccPackedRGB16Pel>**.

  Parameters

  buf — A reference to the 16-bit RGB pel buffer into which the image will be copied. Must be presized to the size of the image you want to save.

  layer — If *ccUITablet::eImageLayer*, then only image pixels are copied. If *ccUITablet::eOverlayLayer*, then only graphics from the overlay layer are copied. If *ccUITablet::eUnspecifiedLayer*, then both image and graphics are copied.

  Notes

  Use this overload if your desktop is set to 16 bits.

  Throws

  OverlayLayerNotEnabled — The overlay layer is not enabled and the specified layer is *ccUITablet::eOverlayLayer*.

- `void getDisplayedImage(ccPelBuffer<ccPackedRGB32Pel>& buf, ccUITablet::Layers layer=ccUITablet::eUnspecifiedLayer);`

  Places the image currently being displayed on this **ccWin32Display** into the supplied **ccPelBuffer<ccPackedRGB32Pel>**. By default, both the image and any overlay graphics are copied.

  Parameters

  buf — A reference to the 32-bit RGB pel buffer into which the image will be copied. Must be presized to the size of the image you want to save.

  layer — If *ccUITablet::eImageLayer*, then only image pixels are copied. If *ccUITablet::eOverlayLayer*, then only graphics from the overlay layer are copied. If *ccUITablet::eUnspecifiedLayer*, then both image and graphics are copied.

  Notes

  Use this overload if your desktop is set to 32 bits.

  Throws

  OverlayLayerNotEnabled — The overlay layer is not enabled and the specified layer is *ccUITablet::eOverlayLayer*.

getPassThroughValue

```
virtual void getPassThroughValue(c_UInt8 &passval) const;

virtual void getPassThroughValue(
  ccPackedRGB16Pel &passval) const;

virtual void getPassThroughValue(
  ccPackedRGB32Pel &passval) const;
```

Retrieves the pass-through value used for overlay display buffers. The pass-through value is the pixel value in the overlay layer that allows pixels in the underlying image layer to show through. These are Win32 implementations of methods inherited from the **ccDisplay** base class.

- `virtual void getPassThroughValue(c_UInt8 &passval) const;`

  Gets the pass-through value used for overlay display buffers.

  Parameters

  passval — Return parameter for the pass-through color used for overlay display buffers on 8-bit displays.

- ```
  virtual void getPassThroughValue(
    ccPackedRGB16Pel &passval) const;
  ```

  Gets the pass-through value used for overlay display buffers.

  Parameters

  passval — Return parameter for the pass-through color used for overlay display buffers on 16-bit displays.

- ```
  virtual void getPassThroughValue(
    ccPackedRGB32Pel &passval) const;
  ```

  Gets the pass-through value used for overlay display buffers.

  Parameters

  passval — Return parameter for the pass-through color used for overlay display buffers on 32-bit displays.

multiSelectKey

```
void multiSelectKey(ccKeyBoardEvent::VirtualKey k);
ccKeyBoardEvent::VirtualKey multiSelectKey() const;
```

- `void multiSelectKey(ccKeyBoardEvent::VirtualKey k);`

Sets the virtual key used for multiple selections. When the user holds this key down while clicking the mouse, multiple items are selected instead of just one. Valid keys are *<Shift>*, *<Ctrl>*, or none. The default is the *<Shift>* key.

Parameters

k The virtual key that enables multiple selections. Must be one of:

ccKeyBoardEvent::eShift
ccKeyboardEvent::eControl
ccKeyboardEvent::eNoVKey

Throws

ccUIError::BadKeyValue
The input is not one of the three valid values.

Notes

As each key can have only one function assigned to it at a time, the setting for **ccWin32Display::multiSelectKey()** interacts with that for **ccWin32Display::panKey()**. If you change the multiselect key from *ccKeyboardEvent::eShift* to *ccKeyboardEvent::eControl*, the pan key is assigned the value *ccKeyboardEvent::eNoVKey*. You can change this with a subsequent call to **ccWin32Display::panKey()**.

- `ccKeyBoardEvent::VirtualKey multiSelectKey() const;`

Retrieves the virtual key used for multiple selections.

panKey

```
void panKey(ccKeyboardEvent::VirtualKey k);
ccKeyboardEvent::VirtualKey panKey() const;
```

- `void panKey(ccKeyboardEvent::VirtualKey k);`

Sets the key used to activate panning with the mouse. When the user holds down the specified key, the mouse cursor changes to a hand so that the image can be panned. The pan key can be either *<Ctrl>, <Shift>*, or none. The *<Ctrl>* key is the default.

Parameters

k — The virtual key that enables panning. Must be one of:

ccKeyboardEvent::eShift
ccKeyboardEvent::eControl
ccKeyboardEvent::eNoVKey

Throws

ccUIError::BadKeyValue
The input is not one of the three valid values.

Notes

As each key can have only one function assigned to it at a time, the setting for **ccWin32Display::panKey()** interacts with that for **ccWin32Display::multiSelectKey()**. If you change the pan key from to *ccKeyboardEvent::eControl* to *ccKeyboardEvent::eShift*, the multiselect key is assigned the value *ccKeyboardEvent::eNoVKey*. You can change this with a subsequent call to **ccWin32Display::multiSelectKey()**.

- `ccKeyboardEvent::VirtualKey panKey() const;`

  Retrieves the virtual key that activates panning.

waitForVerticalBlank

```
void waitForVerticalBlank(bool set);
```

Specifies whether to synchronize the image display to the vertical blank of the monitor. Not all host platforms support this feature with reasonable CPU usage.

Parameters

set — If true, the display is synchronized to the vertical blank of the monitor. If false, it is not synchronized.

activeLockedScope

```
ccWin32UIScope* activeLockedScope() const;
```

For Cognex internal use only.

Static Functions

platformCompatibility

```
static bool platformCompatibility();
```

For Cognex internal use only.

Deprecated Members

The following **ccWin32Display** methods are all deprecated.

chromaKey

```
void chromaKey(COLORREF color);
COLORREF chromaKey() const;
```

chromaKeyingEnabled

```
bool chromaKeyingEnabled() const;
```

enableChromaKeying

```
void enableChromaKeying(bool enable);
```

getDC

```
HDC getDC();
```

releaseDC

```
void releaseDC(HDC dc);
```

ccWorkerThreadManager

```
#include <ch_cvl/workmgr.h>

class ccWorkerThreadManager;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	No

ccWorkerThreadManager implements three static functions that let you control the system-wide creation of worker threads by multi-core-optimized image processing and vision tools.

Public Member Functions

configure

```
static void configure(
   const ccWorkerThreadManagerParams &params);
```

Applies the worker thread creation settings defined in the supplied **ccWorkerThreadManagerParams** object to this system. The settings apply to the currently running application

Parameters

params The thread creation parameters to apply.

Notes

This function does not affect any already-running worker threads created by running CVL tools. If you call this function to change the number of allowed worker threads, then the original threads will complete normally, but may contend for resources with the newly created worker threads until they complete their work.

activateMax

```
static void activateMax();
```

A convenience function that enables the creation of a worker thread for each core detected on the current system.

Notes

This function does not affect any already-running worker threads created by running CVL tools. If you call this function, all currently running worker threads will complete normally, but may contend for resources with the newly created worker threads until they complete their work.

deactivate `static void deactivate();`

Disables the creation of worker threads.

ccWorkerThreadManagerDefs

```
#include <ch_cvl/workmgr.h>

class ccWorkerThreadManagerDefs;
```

Enumerations

DesiredWorkerThreads

```
enum DesiredWorkerThreads;
```

This enumeration specifies number of worker threads created by multi-processor-optimized vision and image processing tools.

Value	Meaning
eWorkerThreadsNone	Do not create any worker threads.
eWorkerThreadsMax	Create one worker thread for each core.
eWorkerThreadsSpecified	Create a specific number of worker threads.
kDefaultWorkerThreads	The default value for this enumeration.

ccWorkerThreadManagerDefs

ccWorkerThreadManagerParams

```
#include <ch_cvlworkmgr.h>

class ccWorkerThreadManagerParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

ccWorkerThreadManagerParams encapsulates the parameters for configuring the Work Manager, which lets you control the number of worker threads created by multi-core-optimized vision and image processing tools.

Constructors/Destructors

ccWorkerThreadManagerParams

```
ccWorkerThreadManagerParams();
```

Constructs a **ccWorkerThreadManagerParams** object with the following default values:

- **desiredWorkerThreads** set to *ccWorkerThreadManagerDefs::eWorkerThreadsMax*
- **workerThreadCount** set to 1

Public Member Functions

desiredWorkerThreads

```
ccWorkerThreadManagerDefs::DesiredWorkerThreads
   desiredWorkerThreads() const;

void desiredWorkerThreads(
   ccWorkerThreadManagerDefs::DesiredWorkerThreads
   workerThreads);
```

- `ccWorkerThreadManagerDefs::DesiredWorkerThreads desiredWorkerThreads() const;`

  Returns the number of worker threads that will be created by multi-core-optimized vision and image processing tools. The returned value is one of the following:

  ccWorkerThreadManagerDefs::eWorkerThreadsNone
  ccWorkerThreadManagerDefs::eWorkerThreadsMax
  ccWorkerThreadManagerDefs::eWorkerThreadsSpecified

- `void desiredWorkerThreads( ccWorkerThreadManagerDefs::DesiredWorkerThreads workerThreads);`

  Sets the number of worker threads that will be created by multi-core-optimized vision and image processing tools. You can specify that no threads be created, that a discrete number of threads (controlled by **workerThreadCount()**) be created, or that one thread be created for every core found on this system (the default).

  Parameters

  workerThreads The number of worker threads to create. *workerThreads* must be one of the following values:

  ccWorkerThreadManagerDefs::eWorkerThreadsNone
  ccWorkerThreadManagerDefs::eWorkerThreadsMax
  ccWorkerThreadManagerDefs::eWorkerThreadsSpecified

  Throws

  ccWorkerThreadManagerDefs::BadParams
  workerThreads is not a member of the **ccWorkerThreadManagerDefs::DesiredWorkerThreads** enumeration.

workerThreadCount

```
c_Int32 workerThreadCount() const;

void workerThreadCount(c_Int32 workerThreadCount);
```

- `c_Int32 workerThreadCount() const;`

  Returns the number of worker threads to create if **desiredWorkerThreads()** is *ccWorkerThreadManagerDefs::eWorkerThreadsSpecified*.

- `void workerThreadCount(c_Int32 workerThreadCount);`

  Sets the number of worker threads to create if **desiredWorkerThreads()** is *ccWorkerThreadManagerDefs::eWorkerThreadsSpecified*. This value is ignored for other values of **desiredWorkerThreads()**.

  The default value is 1.

  Parameters

  workerThreadCount
  The number of worker threads to create.

  Notes

  This function does not compare the supplied value against the actual number of cores on the current system. Specifying a value that is greater than the actual number of cores can reduce the performance of your application.

  Throws

  ccWorkerThreadManagerDefs::BadParams
  workerThreadCount is less than or equal to 0.

ccWorkerThreadManagerParams

cc_FeatureRange

```
#include <ch_cvl/pmifproc.h>

class cc_FeatureRange;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	Simple

This class describes a range of boundary points within a PatInspect feature. You use the member functions of this class to obtain information about **ccFeatureSegment**s and **ccMouseBite**s, both of which are derived from this class.

Note You should not instantiate this class directly.

Constructors/Destructors

cc_FeatureRange

```
cc_FeatureRange();
```

Constructs a **cc_FeatureRange**.

Public Member Functions

startPos

```
int startPos() const;
```

Return the index of the first boundary point in this range of boundary points.

numPoints

```
int numPoints() const;
```

Return the number of boundary points in this feature range.

cc_FeatureRange

cc_PelBuffer

```
#include <ch_cvl/pelbuf.h>

class cc_PelBuffer;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This is an abstract class used to define **ccPelBuffer** and **ccPelBuffer_const**. It describes type-independent attributes of windows on root images.

Note This chapter uses the term “window” to describe a rectangular region of a root image. The term “pel buffer” is also used to describe the same thing.

Constructors/Destructors

The constructors and destructors of this class are used only by Cognex-supplied derived classes **ccPelBuffer** and **ccPelBuffer_const**.

Operators

These operators are available to the Cognex-supplied derived classes **ccPelBuffer** and **ccPelBuffer_const**.

operator=

```
cc_PelBuffer& operator= (const cc_PelBuffer& pelbuf);
```

Makes this window have the same root image, root offset, offset, size, and transform as *pelbuf*.

Parameters

pelbuf The window whose attributes you wish to copy.

Notes

If *pelbuf* is unbound, this window becomes unbound.

operator== `bool operator== (const cc_PelBuffer& pelbuf) const;`

Returns true if this window and *pelbuf* have the same root image and have the same size, offset and root offset. The transforms of the two pel buffers are not compared.

Parameters

pelbuf The window to compare.

Notes

If both pel buffers are unbound, but have the same offset, root offset and size, this function returns true.

operator!= `bool operator!= (const cc_PelBuffer& pelbuf) const;`

Returns true if this window and *pelbuf* do not have the same root image, the same size, the same offset, and the same root offset. The transforms of the two pel buffers are not compared.

Parameters

pelbuf The window to compare.

Public Member Functions

bind `bool bind(const cc_PelBuffer& pelbuf, bool clip = false);`

Makes this window have the same root image as *pelbuf*. *Pelbuf*'s root offset, offset, transform, and size are ignored. If *pelbuf* is unbound, this window becomes unbound. Returns true if the result was clipped to fit within the new image; false otherwise.

This window's root offset, offset and size are not modified unless clipping is required. If *clip* is true, the result is clipped to the extent of the new root image (possibly resulting in a null window result if the window's location is entirely outside of the new root image). This window's transform is not changed.

Parameters

pelbuf The window whose root image you want to bind this window to.

clip If true, clip to the extent of the new root image.

Throws

ccPel::BadWindow

The window's location does not fit entirely within this window's new root image, and *clip* is false.

isBound `bool isBound() const;`

Return true if this window is bound to a root image.

setUnbound `void setUnbound();`

Make this window unbound.

Notes

An unbound window retains its root-offset, offset and size information. See **bind()**.

width `c_Int32 width() const;`

Returns the width of this window, in pixels.

height `c_Int32 height() const;`

Returns the height of this window, in rows.

size `ccIPair size() const;`

Returns a pair containing this window's width as the x component, and its height as the y component.

rowUpdate `c_Int32 rowUpdate() const;`

Returns the value, in pixels, to be added to a pixel pointer to move the pointer to the next row, staying in the same column.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

Notes

rowUpdate() will be different from the **width()** if the window does not span the width of the entire root image or if the root image is padded.

alignModulus `c_Int32 alignModulus() const;`

Returns the alignment modulus that was used when constructing this window's root image. See **ccPelRoot** on page 2391.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

root `cc_PelRoot* root() const;`

Returns a pointer to this window's root image.

Throws

ccPel::UnboundWindow
This window is not bound to any root image.

Notes

Do not cache the returned pointer.

disconnectRoot

```
cc_PelRoot* disconnectRoot();
```

Disconnects this window from its root image, causing the window to become unbound, but does not delete root image. Returns the disconnected root image.

sharesPels

```
bool sharesPels(const cc_PelBuffer& pelbuf) const;
```

Returns true if this window and *pelbuf* have same root image and if they have one or more pixels in common.

Parameters

pelbuf The window to compare with this window.

Notes

If this window or *pelbuf* are unbound, this function returns false. The offset has no effect on this operation.

subWindow

```
void subWindow(c_Int32 x, c_Int32 y, c_Int32 width,
  c_Int32 height);

void subWindow(const ccIPair& origin, const ccIPair& size);
```

- ```
  void subWindow(c_Int32 x, c_Int32 y, c_Int32 width,
    c_Int32 height);
  ```

Makes this window a subwindow of itself. *X* and *y* are added to the window's origin (the offset) before the change. The window's transform is not changed.

Parameters

x The amount to add to the x-coordinate of the window's offset.

y The amount to add to the y-coordinate of the window's offset.

width The new width of the window.

height The new height of the window.

Throws
  ```

ccPel::BadWindow
The new subwindow does not fit entirely within this window. This error is also raised if width or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

- `void subWindow(const ccIPair& origin, const ccIPair& size);`

Makes this window a subwindow of itself. *Origin* is relative to the window's origin (the offset) before the change. The window's transform is not changed.

Parameters

origin The x and y amounts to add to the window's current offset.

size The width (x component) and height (y component) of the subwindow.

Throws

ccPel::BadWindow
The new subwindow does not fit entirely within this window. This error is also raised if width or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

window

```
ccPelRect window() const;

bool window(c_Int32 x, c_Int32 y, c_Int32 width,
  c_Int32 height, bool clip=false);

bool window(const ccIPair& origin, const ccIPair& size,
  bool clip=false);

bool window(const ccPelRect& region, bool clip=false);
```

- `ccPelRect window() const;`

Returns the window's rectangle. The top, left coordinate of this rectangle is the window's offset. The size is the window's size.

- ```
  bool window(c_Int32 x, c_Int32 y, c_Int32 width,
    c_Int32 height, bool clip=false);
  ```

  Changes the location and dimensions of this window. *X* and *y* are given in the window's coordinate system, that is relative to the window's offset. Returns true if the window was clipped to fit the root image; false otherwise.

  If *clip* is true, the resulting window is clipped to root image, possibly resulting in a null window if the new location is entirely outside the root image. The window's transform is not changed.

  Parameters

x	The new x-coordinate of the window's origin, in window coordinates.
y	The new y-coordinate of the window's origin, in window coordinates.
width	The new width of the window in pixels.
height	The new height of the window in rows.
clip	If true, the window is clipped to the extent of the root image.

  Throws

  ccPel::BadWindow
  The requested window does not fit entirely within this window's root image, and clip is false. This error is also raised if width and/or height are negative.

  ccPel::UnboundWindow
  This window is not bound to any root image.

- ```
  bool window(const ccIPair& origin, const ccIPair& size,
    bool clip=false);
  ```

  Changes the location and dimensions of this window. Returns true if the window was clipped to fit the root image; false otherwise.

  Origin is given in the window's coordinate system, that is relative to the window's offset. If *clip* is true, the resulting window is clipped to root image, possibly resulting in a null window if the new location is entirely outside the root image. The window's transform is not changed.

  Parameters

origin	The new origin of the window in window coordinates.
size	The new size of the window.

clip If true, the window is clipped to the extent of the root image.

Throws

ccPel::BadWindow
The requested window does not fit entirely within this window's root image, and clip is false. This error is also raised if width and/or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

- `void window(const ccPelRect& region, bool clip=false);`

Changes the location and dimensions of this window. Returns true if the window was clipped to fit the root image; false otherwise.

Region is given in the window's coordinate system, that is relative to the window's offset. If *clip* is true, the resulting window is clipped to root image, possibly resulting in a null window if the new location is entirely outside the root image. The window's transform is not changed.

Parameters

region A rectangle, in window coordinates, that describes the new location of the window.

clip If true, the window is clipped to the extent of the root image.

Throws

ccPel::BadWindow
The requested window does not fit entirely within this window's root image, and clip is false. This error is also raised if width and/or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

windowRoot

```
ccPelRect windowRoot() const;

bool windowRoot(c_Int32 x, c_Int32 y, c_Int32 width,
  c_Int32 height, bool clip=false);

bool windowRoot(const ccIPair& origin, const ccIPair& size,
  bool clip=false);

bool windowRoot(const ccPelRect& region, bool clip=false);
```

- `ccPelRect windowRoot() const;`

  Returns the window's position in the root image. This function returns a rectangle whose top, left coordinate is the window's root offset and whose size is the window's size.

- `bool windowRoot(c_Int32 x, c_Int32 y, c_Int32 width, c_Int32 height, bool clip=false);`

  Changes the location and dimensions of this window within the root image. Returns true if the window was clipped to fit the root image; false otherwise.

  X and *y* are given in the root image's coordinate system. The window's offset (its origin) is adjusted by the change in the root offset.

  If *clip* is true, the resulting window is clipped to root image, possibly resulting in a null window if the new location is entirely outside the root image. The window's transform is not changed.

  Parameters

x	The new x-coordinate of the window's root offset, in root image coordinates.
y	The new y-coordinate of the window's root offset, in root image coordinates.
width	The new width of the window in pixels.
height	The new height of the window in rows.
clip	If true, the window is clipped to the extent of the root image.

  Throws

  ccPel::BadWindow

  The requested window does not fit entirely within this window's root image, and clip is false. This error is also raised if width and/or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

- `bool windowRoot(const ccIPair& origin, const ccIPair& size, bool clip=false);`

Changes the location and dimensions of this window within the root image. Returns true if the window was clipped to fit the root image; false otherwise.

Origin is given in the root image's coordinate system. The window's offset (its origin) is adjusted by the change in the root offset.

If *clip* is true, the resulting window is clipped to root image, possibly resulting in a null window if the new location is entirely outside the root image. The window's transform is not changed.

Parameters

origin The new origin of the window in root image coordinates.

size The new size of the window.

clip If true, the window is clipped to the extent of the root image.

Throws

ccPel::BadWindow
The requested window does not fit entirely within this window's root image, and clip is false. This error is also raised if width and/or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

- `bool windowRoot(const ccPelRect& region, bool clip=false);`

Changes the location and dimensions of this window within the root image. Returns true if the window was clipped to fit the root image; false otherwise

Region is given in the root image's coordinate system. The window's offset (its origin) is adjusted by the change in the root offset.

If *clip* is true, the resulting window is clipped to root image, possibly resulting in a null window if the new location is entirely outside the root image. The window's transform is not changed.

Parameters

region A rectangle, in root image coordinates, that describes the new location of the window.

clip If true, the window is clipped to the extent of the root image.

Throws

ccPel::BadWindow
The requested window does not fit entirely within this window's root image, and clip is false. This error is also raised if width and/or height are negative.

ccPel::UnboundWindow
This window is not bound to any root image.

offsetRoot

```
ccIPair offsetRoot() const;
```

Returns the root offset of this window relative to its root image origin. The root offset is the distance from the top, left corner of the root image to the top, left corner of the window. The top, left coordinate of the root image is always (0,0).

offset

```
ccIPair offset() const;
void offset(c_Int32 left, c_Int32 top);
void offset(const ccIPair& origin);
```

- `ccIPair offset() const;`

Returns the window's offset, a pair that gives the image coordinates of this window's top, left corner. The offset allows you to provide an arbitrary coordinate system for the window independent of the root image's coordinates.

- `void offset(c_Int32 left, c_Int32 top);`

Sets the origin (the offset) of the window to *(left, top).* Note that this function changes the window's coordinate system and does not change the location of the window with respect to the root image. The window's transform is not changed.

Parameters

left The x-coordinate of the window's origin.

top The y-coordinate of the window's origin.

- `void offset(const ccIPair& origin);`

Sets the origin (the offset) of the window to *origin.* Note that this function changes the window's coordinate system and does not change the location of the window with respect to the root image. The window's transform is not changed.

Parameters

origin The coordinates of the window's origin.

copyXforms

```
void copyXforms(const cc_PelBuffer& win);
```

Sets this window's transformation object to be the same as another window's transformation object. This function is the most efficient way to make to windows (pel buffers) use the same transformation object.

Parameters

win The window that has the transformation object to copy.

clientFromImageXformBase

```
cc2XformBasePtrh_const clientFromImageXformBase() const;

void clientFromImageXformBase(
  const cc2XformBasePtrh_const& xform, bool copy = true);

void clientFromImageXformBase(const cc2XformBase& xform);
```

- `cc2XformBasePtrh_const clientFromImageXformBase() const;`

  Returns this window's transformation object such that it is suitable for transforming the window's image coordinates to the client's reference coordinates.

  Example

  To get the client coordinates of the center of the top, left pixel in this window, you would use the following:

  ```
  cc2Vect clientPt = (*clientFromImageXformBase()) *
                  cc2Vect(offset().x + 0.5), offset().y + 0.5);
  ```

  To transform an arbitrary location (x, y) in this window's image coordinates to client coordinates, you would use the following:

  ```
  cc2Vect clientPt = (*clientFromImageXformBase()) *
                      cc2Vect(x,y);
  ```

- ```
  void clientFromImageXformBase(
    const cc2XformBasePtrh_const& xform, bool copy = true);
  ```

  Sets this window's transformation object suitable for transforming the window's image coordinates to the client's reference coordinates. Setting a new transformation does not affect the window's offset.
  ```

Parameters

xform	A pointer handle to the transformation object.
copy	If true, this window is assigned a copy of *xform*. If false, this window is assigned a reference to *xform*.

Notes

If *copy* is false, you must ensure that *xform* will not change during this window's lifetime.

- `void clientFromImageXformBase(const cc2XformBase& xform);`

Sets this window's transformation object suitable for transforming the window's image coordinates to the client's reference coordinates. Setting a new transformation does not affect the window's offset.

Parameters

xform	A transformation object.

Notes

This setter always makes a copy of *xform*.

Both **imageFromClientXformBase()** and **clientFromImageXformBase()** setters set the same object. Use one or the other, but not both. The deprecated functions **imageFromClientXform()** and **clientFromImageXform()** setters also operate on the same object.

imageFromClientXformBase

```
cc2XformBasePtrh_const imageFromClientXformBase() const;

void imageFromClientXformBase(
  const cc2XformBasePtrh_const& xform, bool copy = true);

void imageFromClientXformBase(const cc2XformBase& xform);
```

- `cc2XformBasePtrh_const imageFromClientXformBase() const;`

Returns this window's transformation object suitable for transforming the client's reference coordinates to the window's image coordinates.

- ```
  void imageFromClientXformBase(
    const cc2XformBasePtrh_const& xform, bool copy = true);
  ```

  Sets this window's transformation object suitable for transforming the client's reference coordinates to the window's image coordinates. Setting a new transformation does not affect the window's offset.

  Parameters

  xform A pointer handle to the transformation object.

  copy If true, this window is assigned a copy of *xform*. If false, this window is assigned a reference to *xform*.

  Notes

  If *copy* is false, you must ensure that *xform* will not change during this window's lifetime.

- ```
  void imageFromClientXformBase(const cc2XformBase& xform);
  ```

  Sets this window's transformation object suitable for transforming the client's reference coordinates to the window's image coordinates. Setting a new transformation does not affect the window's offset.

  Parameters

  xform A transformation object.

  Notes

  This setter always makes a copy of *xform*.

  Both **imageFromClientXformBase()** and **clientFromImageXformBase()** setters set the same object. Use one or the other, but not both. The deprecated functions **imageFromClientXform()** and **clientFromImageXform()** setters also operate on the same object.

clientFromImageXform

```
cc2Xform clientFromImageXform() const;

void clientFromImageXform(const cc2Xform& xform);
```

- ```
  cc2Xform clientFromImageXform() const;
  ```

  Returns this window's 6-degree-of-freedom transformation object suitable for transforming the window's image coordinates to the client's reference coordinates.
  ```

Example

To get the client coordinates of the center of the top, left pixel in this window, you would use the following:

```
cc2Vect clientPt = clientFromImageXform() *
                   cc2Vect(offset().x + 0.5), offset().y + 0.5);
```

To transform an arbitrary location (x, y) in this window's image coordinates to client coordinates, you would use the following:

```
cc2Vect clientPt = clientFromImageXform() * cc2Vect(x,y);
```

- `void clientFromImageXform(const cc2Xform& xform);`

Sets this window's 6-degree-of-freedom transformation object suitable for transforming the window's image coordinates to the client's reference coordinates. Setting a new transformation does not affect the window's offset.

Parameters

xform The new transformation object for the window.

Notes

Both **imageFromClientXform()** and **clientFromImageXform()** setters, as well as the **imageFromClientXformBase()** and **clientFromImageXformBase()** setters, set the same object.

imageFromClientXform

```
cc2Xform imageFromClientXform() const;

void imageFromClientXform(const cc2Xform& xform);
```

- `cc2Xform imageFromClientXform() const;`

Returns this window's 6-degree-of-freedom transformation object suitable for transforming the client's reference coordinates to the window's image coordinates.

- `void imageFromClientXform(const cc2Xform& xform);`

Sets this window's 6-degree-of-freedom transformation object suitable for transforming the client's reference coordinates to the window's coordinates. Setting a new transformation does not affect the window's offset.

Parameters

xform The new transformation object for the window.

Notes

Both **imageFromClientXform()** and **clientFromImageXform()** setters, as well as the **imageFromClientXformBase()** and **clientFromImageXformBase()** setters, set the same object.

isEntire

```
bool isEntire() const;
```

Returns true if window encompasses the entire root image.

Notes

If this window is unbound, returns false.

setEntire

```
void setEntire();
```

Sets the window's offset (origin) and size to encompass the entire root image. The window's offset (origin) is modified appropriately.

Throws

ccPel::UnboundWindow

This window is not bound to any root image.

windowEntire

```
ccPelRect windowEntire() const;
```

Returns a rectangle object whose origin is the upper left corner of the window's root in image coordinates (offset) and whose size is the size of this window's root.

refc

```
c_Int32 refc() const;
```

Returns the number of windows (including this one) that reference the underlying root image.

cc_PelRoot

```
#include <ch_cvl/pelbuf.h>

class cc_PelRoot : public virtual ccPersistent, public ccRepbase,
  public ccRemoteObject;
```

Class Properties

Copyable	Yes
Derivable	Cognex-supplied classes only
Archiveable	Complex

This class should be considered part of class **ccPelRoot** (see **ccPelRoot** on page 2391). It describes type-independent attributes of root images.

Constructors/Destructors

The constructors for this class are protected. To create a root image, use class **ccPelRoot** instead.

Public Member Functions

width

```
c_Int32 width() const;
```

Returns the width of this root image in pixels. The result does not include any pad pixels.

height

```
c_Int32 height() const;
```

Returns the height of this root image, in rows.

rowUpdate

```
c_Int32 rowUpdate() const;
```

Returns the value, in pixels, to be added to a pixel pointer to move the pointer to the next row, staying in the same column.

Notes

The value returned by **rowUpdate()** may be different from the value returned by **width()** if the root image is padded. The number of pad pixels is **rowUpdate() – width()**.

alignModulus `c_Int32 alignModulus() const;`

Returns the alignment modulus value that was used when constructing this root image **ccPelRoot** on page 2391.

mutating `void mutating();`

Informs the persistence system that this root image's pixel values are about to change or have already changed.

release `void release();`

Mark this root image as no longer in use. Use this function instead of the **delete** operator when you are finished using an root image.

In order to reduce the overhead of creating root images, **cc_PelRoot** maintains a pool of preconstructed **cc_PelRoot** objects. Using the release function rather than the **delete** operator ensures that this root image returns to the pool. If this root image did not come from the pool (because you provided storage for the image), this function does delete this object.

cc_PMDefs

```
#include <ch_cvl/pmpbase.h>

class cc_PMDefs;
```

Note This is an abstract base class used to define enumerations and constants shared by all of the PatMax tools.

Enumerations

DOF `enum DOF`

This enumeration defines the degrees of freedom supported by PatMax.

Value	Meaning
kAngle = 1	Angle
kUniformScale = 2	Uniform scale
kXScale = 4	x-scale
kYScale = 8	y-scale

Algorithm `enum Algorithm`

This enumeration defines the algorithms supported by PatMax.

Value	Meaning
kPatquick = 1	Use the faster, less accurate PatQuick algorithm.
kPatmax = 2	Use the slower, more accurate PatMax algorithm.
kPatflex = 4	Use the PatFlex deformation algorithm.
kPatpersp = 16	Use the PatPersp deformation algorithm.

Refinement

`enum Refinement`

Algorithms for deformation refinement, used by PatFlex only.

Value	Meaning
kRefinementNone = 0	No refinement. There may be small errors in the transform's mapping.
kRefinementCoarse = 1	The transform will be refined to the accuracy limit of the coarse grain limit set for the pattern.
kRefinementMedium = 2	The transform will also be refined to resolve high-level deformation inaccuracies that can occur in some applications.
kRefinementFine = 3	The transform will be refined to the accuracy limit of the fine grain limit set for the pattern.

DeformationFit

`enum DeformationFit`

Transforms for deformation fitting, used by PatFlex only.

Value	Meaning
kFitNone = 0	No transform is computed.
kFitDeformation = 1	Computes a deformation transform. See **cc2XformDeform**.
kFitPerspective = 2	The computed transform includes perspective deformation correction. See **cc2XformPerspective**.

SensitivityMode

`enum SensitivityMode`

This enumeration defines the PatMax/PatQuick sensitivity modes.

Value	Meaning
kSensitivityModeStandard = 1	Normal PatMax/PatQuick mode.
kSensitivityModeHigh = 2	High sensitivity mode.

SaveMatchInfoMode

```
enum SaveMatchInfoMode
```

This enumeration defines the match information saving modes for PatMax.

Value	Meaning
kSaveMatchInfoModeNone = 0x0	No match information is saved.
kSaveMatchInfoModeClassic = 0x1	Match information about the pattern features is saved to allow display only, via **cc_PMResult::displayMatch()**. Match information is not persisted with the result.
kSaveMatchInfoModePattern = 0x2	Match information about the pattern features is saved to allow display via **cc_PMResult::displayMatch()** and also to allow direct access via **cc_PMResult::matchInfo()**. Match information is persisted with the result.
kSaveMatchInfoModeImage = 0x4	Match information about the run-time image features is saved to allow display and direct access via **cc_PMResult::matchInfo()**. Match information is persisted with the result.

Constants

The following constants provide default values for certain training-time and run-time parameters:

Value	Meaning
kDefaultNumToFind = 1	An integer specifying the default number of pattern instances to find.
kDefaultAcceptThresholdTimes1000 = 500	An integer specifying the default acceptance threshold times 1000.
kDefaultContrastThresholdTimes1000 = 10000	An integer specifying the default contrast threshold times 1000.
kDefaultZonesEnabled = 0	An integer specifying the default zones enabled.

Value	Meaning
kDefaultAngleNominal = 0	An integer specifying containing the default nominal angle value.
kDefaultUniformScaleNominal = 1	An integer specifying containing the default nominal uniform scale value.
kDefaultXScaleNominal = 1	An integer specifying containing the default nominal X scale value.
kDefaultYScaleNominal = 1	An integer specifying containing the default nominal Y scale value.
kDefaultXYOverlapTimes1000 = 800	An integer specifying the default area overlap threshold times 1000.
kDefaultUseClutter = 1	An integer specifying the default value for whether or not to consider clutter when scoring pattern instances.

cc_PMInspectFeature

```
#include <ch_cvl/bndpnts.h>

template <class BoundaryPoint>class cc_PMInspectFeature;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A base class for describing PatInspect features.

Note You should not instantiate this class directly.

Constructors/Destructors

cc_PMInspectFeature

```
cc_PMInspectFeature();

virtual ~cc_PMInspectFeature();
```

- `cc_PMInspectFeature();`

  Constructs a **cc_PMInspectFeature**.

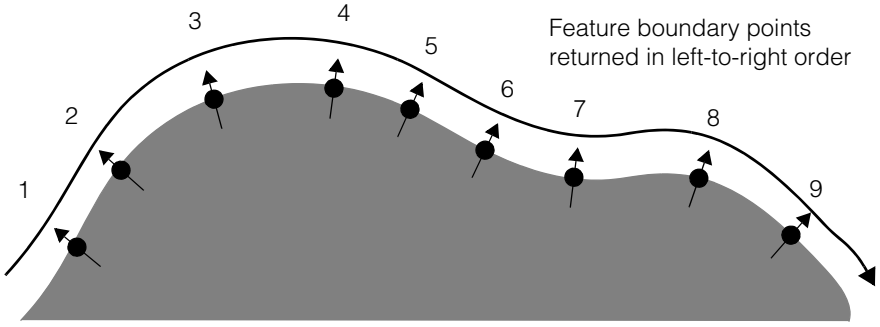
- `virtual ~cc_PMInspectFeature();`

  Destroys a **cc_PMInspectFeature**.

Public Member Functions

boundaryPoints `const cmStd vector<BoundaryPoint>& boundaryPoints() const;`

Return a vector of feature boundary points which specify the feature. The points are returned in left-to-right order based on the gradient direction, as shown in the following figure:



matchQuality `double matchQuality() const;`

Return the mean match quality for all the feature boundary points in this feature.

weight `double weight() const;`

Return mean weight for all the feature boundary points in this feature.

isClosed `bool isClosed() const;`

Returns true if this feature is closed, false if it is open.

cc_PMPattern

```
#include <ch_cvl/pmpbase.h>

class cc_PMPattern;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single trained PatMax pattern. This class includes functions to set the training parameters and a function to search for the pattern in a search image.

This is an abstract class used to define functions shared by all of the PatMax pattern classes. Do not instantiate this class directly.

Public Member Functions

trainClientFromPattern

```
cc2Xform trainClientFromPattern() const;

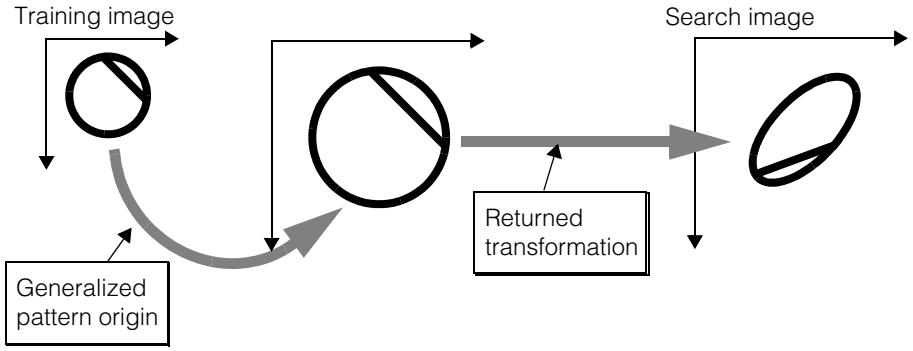
virtual void trainClientFromPattern(
  const cc2Xform& genOrigin);
```

- `cc2Xform trainClientFromPattern() const;`

  Retrieves the mapping from pattern coordinates to training image client coordinates.

- `virtual void trainClientFromPattern(`
  `const cc2Xform& genOrigin);`

Sets the mapping from pattern coordinates to training image client coordinates.:



Parameters

transform The new mapping from pattern coordinates to training image client coordinates. The default is the identity transform.

Notes

Pattern coordinates are a client-controlled coordinate system that specifies how PatMax should match the pattern to an image and report result poses. A result pose is a mapping from pattern coordinates to run-time client coordinates. Likewise, the generalized degrees of freedom (DOFs) define the search range as a mapping from pattern coordinates to run-time client coordinates. Pattern coordinates may be set in all six degrees of freedom, and can be set and changed at will before and after training. Pattern coordinates include the pattern's origin as its translation component. Setting the pattern coordinates with this member always sets the origin.

Training never modifies the pattern coordinates, which are relative to the client coordinates established by the most recent training, if any has taken place.

---

origin

```
cc2Vect origin() const;

virtual void origin(const cc2Vect& theOrigin);
```

- `cc2Vect origin() const;`

Retrieves the pattern origin in the client coordinate system of the training image. The origin is the translation component of the pattern coordinate system (see **trainClientFromPattern()**). The origin is a point that is attached to the pattern, but may reside outside of the pattern. When PatMax finds an instance of the pattern, the position of the origin in the search image defines the result location.

- `virtual void origin(const cc2Vect& theOrigin);`

Sets the pattern origin. *theOrigin* must be specified in the client coordinate system of the pattern training image. PatMax returns the location of patterns in search images in terms of the location of the pattern origin in the search image client coordinate system.

Parameters

theOrigin The origin. The origin can be located outside the pattern training image. The default origin is (0,0).

Notes

Calling **trainClientFromPattern()** changes the origin. Setting the origin changes the translation component of the generalized pattern origin, but not any of the geometric components (for example, the matrix portion of the transform).

The origin can be read or written for either trained or untrained patterns. Changing the origin does not require the pattern to be retrained.

Setting the origin affects the location of the result as well as the pose.

ignorePolarity

```
virtual bool ignorePolarity() const;
virtual void ignorePolarity(bool thePolarity);
```

- `virtual bool ignorePolarity() const;`

Returns the state of the *ignorePolarity* flag.

- `virtual void ignorePolarity(bool thePolarity);`

Specifies whether PatMax should consider the polarity of features to be significant.

Parameters

thePolarity If true, PatMax will ignore polarity of features. If *false* (the default), only patterns whose features have matching polarity are found.

Notes

If this pattern is currently trained, its state is updated immediately.

elasticity

```
double elasticity() const;
void elasticity(double e);
```

- `double elasticity() const;`

Returns this **cc_PMPattern**'s elasticity value.

- `void elasticity(double e);`

  Sets this **cc_PMPattern**'s elasticity value. The elasticity value is the amount of feature variance, in pixels, that the tool allows.

  Setting a nonzero elasticity value permits the tool to find pattern instances with nonlinear geometric variation to be found.

  Parameters

  e The elasticity value in pixels.

  Throws

  cc_PMDefs::BadParams

  e is less than 0.

  Notes

  If this pattern is currently trained, its state is updated immediately.

autoSelectGrainLimits

```
bool autoSelectGrainLimits() const;

void autoSelectGrainLimits(bool autoSelect);
```

- `bool autoSelectGrainLimits() const;`

  Returns whether or not this **cc_PMPattern** is configured to automatically select coarse and fine granularity limits for feature detection.

- `void autoSelectGrainLimits(bool autoSelect);`

  Controls whether or not this **cc_PMPattern** automatic selects coarse and fine granularity limits for feature detection at training time.

  Parameters

  autoSelect If true, PatMax selects the granularity limits automatically. If false, PatMax uses the granularity limits you specify by calling **grainLimits()**.

  Notes

  The default is true. Enabling automatic selection generally doubles training time.

coarseGrainLimit

```
float coarseGrainLimit() const;
```

Returns the coarse granularity limit for feature detection. The default is 4.0.

fineGrainLimit `float fineGrainLimit() const;`

Returns the fine granularity limit for feature detection. The default is 1.0.

grainLimits `void grainLimits(float coarse, float fine);`

Sets the coarse and fine granularity limits used for feature detection during pattern training.

The automatically set values are appropriate for almost all applications. For information on granularity limits, see the *CVL Vision Tools Guide*.

Parameters

coarse The coarse granularity limit.

fine The fine granularity limit.

Throws

cc_PMDefs::BadParams
Either *coarse* or *fine* is less than 1.0 or greater than 25.5 or *coarse* is less than *fine*.

Notes

If you call this function after training a pattern, the limits you specify will be used for feature detection in the run-time image.

customize `ccCvlString customize(const ccCvlString& pcp);`

Loads the supplied PatMax Customization Pack (PCP). A PCP is a non-human-readable string which contains object code fragments and configuration data.

Loading a PCP only affects the object into which it is loaded, and only a single PCP can be active for a given object at a given time. Call **train()** after loading a PCP.

This function returns a string containing encrypted diagnostic information for use by Cognex.

Parameters

pcp The PCP to load. If *pcp* is NULL, all internal parameters are reset to their default values.

customizeFromFile

```
ccCvlString customizeFromFile(
  const ccCvlString& filename);
```

Loads the supplied PatMax Customization Pack (PCP) file. A PCP is a non-human-readable string which contains object code fragments and configuration data.

Loading a PCP only affects the object into which it is loaded, and only a single PCP can be active for a given object at a given time. Call **train()** after loading a PCP.

This function returns a string containing encrypted diagnostic information for use by Cognex.

Parameters

filename The PCP file to load. If *pcp* is NULL, all internal parameters are reset to their default values.

customizeString

```
ccCvlString customizeString() const;
```

Returns the last customization string set by either **customize()** or **customizeFromFile()**. If neither has been called, this function returns an empty string.

isTrained

```
bool isTrained() const;
```

Returns true if this **cc_PMPattern** is currently trained, *false* otherwise.

algorithms

```
c_UInt32 algorithms() const;
```

Returns a **c_UInt32** formed by ORing together all the algorithms for which this **cc_PMPattern** is trained. The returned value is formed by ORing together one or more of the following values:

cc_PMDefs::kPatquick
cc_PMDefs::kPatmax
cc_PMDefs::kPatflex
cc_PMDefs::kPatpersp

Scope the enumerations for the algorithm using the appropriate name space class such as **cc_PMDefs**.

Throws

cc_PMDefs::NotTrained
If this pattern is not trained for any algorithms.

trainClientFromImage

```
const cc2XformBasePtrh_const trainClientFromImage() const;
```

Returns the client from image transform with which this pattern was trained.

Throws

cc_PMDefs::NotTrained
This pattern is not trained.

---

sensitivityMode

```
cc_PMDefs::SensitivityMode sensitivityMode() const;

void sensitivityMode(cc_PMDefs::SensitivityMode);
```

Controls the sensitivity mode used by PatQuick and PatMax. The sensitivity mode can be either *standard*, or *high*. High sensitivity mode is designed to improve PatQuick and PatMax performance for noisy images and/or patterns with extremely low contrast. PatQuick and PatMax will run slower in high sensitivity mode than in standard sensitivity mode. The default is *cc_PMDefs::kSensitivityModeStandard*.

When using high sensitivity mode, you can control the sensitivity with the sensitivity parameter. See **sensitivityParameter()** below.

Sensitivity mode is added in CVL 6.2. *Standard* sensitivity mode corresponds to PatQuick/PatMax behavior in releases prior to CVL 6.2.

Notes

ccPMInspectPattern derives from **cc_PMPattern** so the sensitivity mode is available to PatInspect as well. However, high sensitivity mode was not designed for use with PatInspect and is not considered useful for that application. Therefore, when using PatInspect always use the default, *cc_PMDefs::kSensitivityModeStandard*.

- `cc_PMDefs::SensitivityMode sensitivityMode() const;`

  Returns the current sensitivity mode.

- `void sensitivityMode(cc_PMDefs::SensitivityMode mode);`

  Sets the sensitivity mode. You should retrain the pattern if the sensitivity mode is changed.

  Parameters

  mode The new sensitivity mode.

  Throws

  cc_PMDefs::BadParams
  If *mode* is not one of the **cc_PMDefs::SensitivityMode** enums.

sensitivityParameter

```
float sensitivityParameter() const;

void sensitivityParameter(float sensitivityParameter);
```

Controls the sensitivity parameter used by PatQuick and PatMax when running in high sensitivity mode. See **sensitivityMode()** above.

The sensitivity parameter must be in the range 1.0 through 10.0. When set to 1.0 PatMax and PatQuick use minimum noise rejection. When set to 10.0 PatMax and PatQuick use maximum noise rejection. Usually the default value 2.0 produces optimal results.

Notes

This is an advanced parameter and should usually be left set to its default value. Before changing **sensitivityParameter** see the *PatMax* chapter of the *Vision Tools Guide* for more information.

- `float sensitivityParameter() const;`

Returns the current sensitivity parameter.

- `void sensitivityParameter(float sensitivityParameter);`

Sets the sensitivity parameter. You should retrain the pattern if you change the sensitivity parameter.

Parameters

sensitivityParameter

The new sensitivity parameter.

Throws

cc_PMDefs::BadParams

If *sensitivityParameter* is outside the range 1.0 through 10.0.

displayFeatures

```
void displayFeatures(ccGraphicList& graphList,
  bool fineGrain, const ccColor& c=ccColor::green)
  const;
```

Draws a feature diagnostic display in the specified color in the supplied **ccGraphicList**. Each feature boundary point detected by PatMax is drawn as a line segment. The line segment is drawn through the center of the feature boundary point and normal to the direction of the feature boundary point.

You can specify that the features detected at the coarsest or finest granularity be drawn.

Parameters

graphList	The graphic list used to draw the feature display.
fineGrain	If true, the features detected at the fine granularity limit are drawn. If false, the features detected at the coarse granularity limit are drawn.
color	The color in which features are drawn. The default is green.

Notes

If this pattern is not trained, no features are drawn.

infoStrings

```
cmStd vector<ccCvlString> infoStrings() const;
```

Returns a vector of training-time diagnostic message strings. For additional information about the diagnostic messages, see the *CVL Vision Tools Guide*.

infoIds

```
cmStd vector<c_UInt32> infoIds() const;
```

Returns a vector of training-time diagnostic message IDs. For a list and description of the diagnostic message IDs, see the *CVL Vision Tools Guide*.

trainTime

```
double trainTime() const;
```

Returns the number of seconds it took to train the pattern.

isAdvancedTrained

```
bool isAdvancedTrained() const;
```

Returns true if this pattern was trained with **ccPMAlignPattern::trainAdvanced()**, false if it was trained using the standard training method (**train()**).

useTrainEdgeThreshold

```
bool useTrainEdgeThreshold() const;

void useTrainEdgeThreshold(bool useTrainEdgeThreshold);
```

- `bool useTrainEdgeThreshold() const;`

  Returns true if PatMax only considers image features with strength above **trainEdgeThreshold()** when training this pattern, false otherwise.

- `void useTrainEdgeThreshold(bool useTrainEdgeThreshold);`

  Sets whether or not to use the edge threshold defined by **trainEdgeThreshold()** when training this pattern. If set to true, the direction of edges with strength less than the threshold are not considered during pattern training. If false, PatMax's dynamically computed edge threshold is used.

  The default value is false.

  Parameters

  useTrainEdgeThreshold

  true to use the **trainEdgeThreshold()** value, false to use the dynamically computed value.

  Notes

  The **trainEdgeThreshold()** value is only used during training. A separate run-time edge threshold, **cc_PMRunParams::edgeThreshold()**, is available.

  trainEdgeThreshold() is an advanced parameter. In most cases, the dynamically computed threshold should be used.

trainEdgeThreshold

```
double trainEdgeThreshold() const;

void trainEdgeThreshold(double trainEdgeThreshold);
```

- `double trainEdgeThreshold() const;`

  Returns the user-selected training edge threshold.

  Notes

  This function does not return PatMax's internally computed threshold, regardless of the value of **useTrainEdgeThreshold()**.

- `void trainEdgeThreshold(double trainEdgeThreshold);`

  If **useTrainEdgeThreshold()** is true, then the value you supply for this function is used during training-time feature extraction. The orientation of image edges with a strength (defined as the difference in grey-level values across the edge) are randomized. If false, PatMax's dynamically computed training edge threshold is used.

  The supplied value must be in the range from 0.0 through 255.0.

  The default value is 10.0.

Parameters

trainEdgeThreshold

The training edge threshold.

Throws

cc_PMDefs::BadParams

edgeThreshold is less than 0.0 or greater than 255.0.

Notes

The **trainEdgeThreshold()** value is only used during training. A separate run-time edge threshold, **cc_PMRunParams::edgeThreshold()**, is available.

trainEdgeThreshold() is an advanced parameter. In most cases, the dynamically computed threshold should be used.

expectedDeformationRate

```
double expectedDeformationRate() const;

void expectedDeformationRate(double e);
```

An estimated pattern deformation rate used with the PatFlex algorithm.

The default is 0.3.

Notes

The expected deformation rate has no effect unless the PatMax algorithm is *kPatflex*. It has no effect on normal PatMax patterns.

If this pattern is currently trained, changing this value will have no effect unless the pattern is retrained.

- `double expectedDeformationRate() const;`

Returns the expected deformation rate.

- `void expectedDeformationRate(double e);`

Sets the expected deformation rate.

Parameters

e The new expected deformation rate.

Throws

cc_PMDefs::BadParams

If $e < 0$, or $e > 1.0$.

id

```
c_Int32 id() const;
```

Returns an identifier used by Multi-Model classes **ccPMMultiModel** and **ccPMMultiModelResultSet** to identify individual PatMax models.

Notes

If the model is not a component of a Multi-Model, then the id will be zero.

Deprecated Members

The following method was deprecated as of CVL 5.5.1.

displayFeatures

```
void displayFeatures(ccUITablet& tablet, bool fineGrain,
  const ccColor& c=ccColor::green) const;
```

Use the overload that takes a **ccGraphicList** reference as the first argument for new development.

cc_PMResult

```
#include <ch_cvl/pmpbase.h>

class cc_PMResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single PatMax search result.

Note This is an abstract class used to define functions shared by all of the PatMax result classes. You should not instantiate this class directly.

Constructors/Destructors

cc_PMResult

```
cc_PMResult();
```

Constructs a **cc_PMResult** containing no meaningful information.

Public Member Functions

location

```
cc2Vect location() const;
```

For this result, returns the location of the pattern origin in the client coordinate system of the search image.

pose

```
const cc2Xform& pose() const;
```

For this result, returns a **cc2Xform** that transforms points in the client coordinate system (offset by any pattern origin) of the pattern training image to the client coordinate system of the search image. You can use this **cc2Xform** to determine the location of any point in the trained pattern in the search image.

If PatMax is run with PatFlex or PatPersp, it returns a linear transform that is the best approximation over the pattern area.

angle

```
ccDegree angle() const;
```

For this result, returns the angle from the x-axis of the search image client coordinate system to the x-axis of the training image client coordinate system (offset by any pattern origin).

xScale

```
double xScale() const;
```

For this result, returns the x-axis scale difference between the trained pattern and the pattern found in the search image.

yScale

```
double yScale() const;
```

For this result, returns the y-axis scale difference between the trained pattern and the pattern found in the search image.

score

```
double score() const;
```

Returns the score that this result received. The score is between 0.0 and 1.0 with higher values indicating a closer match between the trained pattern and the pattern in the search image.

contrast

```
double contrast() const;
```

Returns the contrast score that this result received. The contrast score is the average difference in grey-level values for all of the pattern features that PatMax matched between the trained pattern and this result.

fitError

```
double fitError() const;
```

Returns a **double** that indicates the degree to which the pattern in the search image conforms to the contour of the trained pattern. The value returned by this function ranges from 0.0, which indicates a perfect fit, to infinity, which indicates a poor fit.

The value returned by **fitError()** does not consider the degree to which parts of the pattern are missing, only the closeness of fit.

Notes

Returns -1 if fit error is not available.

coverage

```
double coverage() const;
```

Returns the coverage score that this result received. The coverage score is the percentage of features in the trained pattern that are also present in this result. The value returned by this function ranges from 0.0 to 1.0.

Notes

Returns -1 if coverage is not available.

clutter

```
double clutter() const;
```

Returns the clutter score that this result received. The clutter score is the number of extraneous features present in this result divided by the number of features in the trained pattern. The value returned by this function ranges from 0.0 to infinity.

Notes

Returns -1 if clutter score is not available.

outsideRegionFeatureProportion

```
double outsideRegionFeatureProportion() const;
```

outsideRegionAreaProportion

```
double outsideRegionAreaProportion() const;
```

Returns useful information when you have used **cc_PMRunParams::outsideRegionThreshold()**. Function **outsideRegionFeatureProportion()** specifies what proportion of the pattern's features fall outside the runtime search region. Function **outsideRegionAreaProportion()** specifies what proportion of the pattern's area falls outside the runtime search region, where area is the rectangular training region mapped by the found pose.

Notes

If **cc_PMRunParams::outsideRegionThreshold()** is not used, then both values return exactly 0.0

Outside region measures are available for PatMax only. The returned values are –1.0 if outside region measures are not available, for example, when the specified algorithm is *kPatflex*.

accepted

```
bool accepted() const;
```

Returns true if the score received by this result is greater than or equal to the score threshold specified for this search.

matchRegion

```
ccAffineRectangle matchRegion() const;
```

Returns a **ccAffineRectangle** which describes the location of the matched pattern in the run-time image's client coordinate system.

imageRegion `ccPelRect imageRegion() const;`

Returns the smallest rectangle in the run-time image's image coordinate system which completely contains the matched pattern.

imageFromClientAtCenter
`const cc2Xform& imageFromClientAtCenter() const;`

Returns a **cc2Xform** that describes the transformation from the client coordinates of the run-time image used to produce this result to the image coordinates of the same image, at the center point of this result.

If the client coordinates of the image are linear, this transformation will be identical for all results and will simply be the image's client coordinates. If the client coordinates of the image are nonlinear, the transformation will represent the linearization of these coordinates at the center of this pattern instance.

isFineStage `bool isFineStage() const;`

Returns true if the final result (pose, score, etc.) came from the fine stage of processing;. Returns false if the final result came from the coarse stage of processing. If there was only one processing stage performed, this function returns true.

Usually, results do come from the fine stage which is more accurate. However, under some circumstances PatMax will decide that the coarse stage result is actually the more reliable and will return the coarse result as the final result.

Notes

Results stored under older releases that did not contain this information will have **isFineStage**() set to true when loaded.

coarseStageResult
`const cc_PMStageResult& coarseStageResult() const;`

fineStageResult `const cc_PMStageResult& fineStageResult() const;`

Returns the coarse or fine stage result. The fine stage result is the final stage of alignment and is typically the most accurate, and the coarse stage result is from a coarser and typically less accurate stage of alignment. If only one stage of processing was performed, both of these results will be the same.

infoStrings

```
cmStd vector<ccCvlString> infoStrings() const;
```

Returns a vector of search-time diagnostic message strings for this individual search result. For additional information about the diagnostic messages, see the *Vision Tool Guide*.

If no diagnostic information is available, the function returns a vector with a size of 0.

infoIds

```
cmStd vector<c_UInt32> infoIds() const;
```

Returns a vector of search-time diagnostic message IDs for this individual search result. For a list and description of the diagnostic message IDs, see the *Vision Tool Guide*.

If no diagnostic information is available, the function returns a vector with a size of 0.

displayMatch

```
void displayMatch(ccUITablet& tablet) const;

void displayMatch(ccGraphicList& graphList) const;
```

Creates a graphic display of the pattern features matched in this PatMax/Align search result. In the resulting display, features drawn in red indicate poor matches (corresponding to a coverage score of less than 0.20), features drawn in yellow indicate fair matches (corresponding to a coverage score from 0.20 to 0.67)), and features drawn in green indicate good matches (corresponding to a coverage score from 0.67 to 1.0).

Notes

This function only produces a graphic display if the search that produces the result is performed using the *ccPMAlignDefs::kPatmax* algorithm and the run-time parameters used for the search are configured to generate and save match information.

For example, you configure a **ccPMAlignRunParams** to generate and save match information by calling **ccPMAlignRunParams::saveMatchInfo()** with an argument of true.

Match information is not saved when you archive a **ccPMAlignResult**.

If match info is unavailable, because the PatQuick algorithm was run, **saveMatchInfoModes()** is *kSaveMatchInfoModeNone*, or the result was restored from an archive, this call will do nothing.

If you are running the PatFlex algorithm, the refinement mode must be set to *kRefinementFine* for this method to have an effect.

- `void displayMatch(ccUITablet& tablet) const;`

  Creates a graphic display in a **ccUITablet** object.

  Parameters

  tablet The tablet where the display is created.

- `void displayMatch(ccGraphicList& graphList) const;`

  Creates a graphics display using a **ccGraphicList** object.

  Parameters

  graphList The display in graphics list format.

saveMatchInfoModes

```
c_UInt32 saveMatchInfoModes() const;
```

Returns the types of match data available, as a bitwise-OR of values from **cc_PMDefs::SaveMatchInfoMode**.

matchInfo

```
const ccPMMatchInfo& matchInfo() const;
```

Returns data about how the pattern features matched the run-time image and/or how the run-time image features matched the pattern.

Notes

If the match data is unavailable, an empty object is returned.

flexResult

```
const ccPMFlexResult &flexResult() const;
```

Returns the PatFlex result object. If the algorithm run was not *kPatflex*, there is no PatFlex result and this function returns a default-constructed **ccPMFlexResult** object.

modelId

```
c_Int32 modelId() const;
```

Returns an identifier used by Multi-Model classes **ccPMMultiModel** and **ccPMMultiModelResultSet** to identify individual PatMax models. The id returned here identifies the model that generated this result.

Notes

If the model used to generate this result is not a component of a Multi-Model, then the id will be zero.

perspectiveResult

```
const ccPMPerspectiveResult &perspectiveResult() const;
```

Returns the PatPersp result object. If the algorithm run was not *kPatpersp*, this result will be default constructed.

maxCoarseAcceptThreshold

```
double maxCoarseAcceptThreshold()const;
```

Returns the maximum coarse accept threshold that would allow this result to be found. This is intended as a guide to setting **cc_PMRunParams::coarseAcceptFrac()**.

Deprecated Members

startPose

```
const cc2Xform& startPose() const;
```

Returns the identity transform. This function is deprecated and is maintained for backward compatibility only.

cc_PMResult

cc_PMRunParams

```
#include <ch_cvl/pmpbase.h>

class cc_PMRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains the run-time parameters used to control a PatMax search.

Note This is an abstract base class used to define functions shared by all of the PatMax run-time parameter classes. Do not instantiate this class directly.

Constructors/Destructors

cc_PMRunParams

```
cc_PMRunParams();
```

Constructs a **cc_PMRunParams** using the default values.

Public Member Functions

acceptThreshold

```
double acceptThreshold() const;

void acceptThreshold(double a);
```

- `double acceptThreshold() const;`

  Returns the threshold score configured for this **cc_PMRunParams**.

- `void acceptThreshold(double a);`

  Sets the threshold score for this **cc_PMRunParams**. Only instances of the pattern that receive scores greater than or equal to *thresh* are marked as accepted. Increasing the value of *thresh* reduces the time required to perform a search.

Parameters

a The threshold. Must be in the range 0.0 through 1.0. The default value is 0.5.

Throws

cc_PMDefs::BadParams

a is less than 0.0 or greater than 1.0.

useCoarseAcceptFrac

```
bool useCoarseAcceptFrac() const;

void useCoarseAcceptFrac(bool useCoarseAcceptFrac);
```

- `bool useCoarseAcceptFrac() const;`

  Returns true if **coarseAcceptFrac()** is used to screen coarse results, false otherwise.

- `void useCoarseAcceptFrac(bool useCoarseAcceptFrac);`

  Sets whether or not to use the coarse accept threshold fraction value specified for **coarseAcceptFrac()**. If true, coarse candidates will be screened using the specified fraction of the **acceptThreshold()** value. If false, the internally computed value is used.

  Parameters

  useCoarseAcceptFrac

  true to use the **coarseAcceptFrac()** value, false to use the internal value.

coarseAcceptFrac

```
double coarseAcceptFrac() const;

void coarseAcceptFrac(double coarseAcceptFrac);
```

- `double coarseAcceptFrac() const;`

  Returns the current coarse accept threshold fraction value. If **useCoarseAcceptFrac()** is true, then this value is used as the accept threshold during the coarse search.

- `void coarseAcceptFrac(double coarseAcceptFrac);`

  Sets the current coarse accept threshold fraction value. If **useCoarseAcceptFrac()** is true, then this value is used as the accept threshold during the coarse search.

  Setting this value allows you to control the behavior of PatMax when it searches using coarse granularity. Reducing this value will exclude more potential matches, speeding up PatMax. Increasing this value forces PatMax to consider more candidates, slowing the search but ensuring that valid candidates are not discarded.

  The default is 0.66.

  Parameters

  coarseAcceptFrac
  The coarse accept threshold fraction.

  Throws

  cc_PMDefs::BadParams
  coarseAcceptFrac is less than 0 or greater than 1.

contrastThreshold

```
double contrastThreshold() const;

void contrastThreshold(double contrast);
```

- `double contrastThreshold() const;`

  Returns the contrast threshold of this **cc_PMRunParams**.

- `void contrastThreshold(double contrast);`

  Sets the contrast threshold of this **cc_PMRunParams**. Only pattern instances where the average difference in pixel values across all feature boundaries exceeds the contrast threshold are considered by PatMax. The default contrast is 10.

  Parameters

  contrast The contrast threshold. If you specify 0 for *contrast*, PatMax considers *all* pattern instances.

  Throws

  cc_PMDefs::BadParams
  contrast is less than 0.

useEdgeThreshold

```
bool useEdgeThreshold() const;

void useEdgeThreshold(bool useEdgeThreshold);
```

- `bool useEdgeThreshold() const;`

  Returns true if PatMax only considers image features with strength above **edgeThreshold()**, false otherwise.

- `void useEdgeThreshold(bool useEdgeThreshold);`

  Sets whether or not to use the edge threshold defined by **edgeThreshold()** when extracting features. If set to true, the direction of edges with strength less than the threshold are randomized. If false, PatMax's dynamically computed edge threshold is used.

  The default value is false.

  Parameters

  useEdgeThreshold

  true to use the **edgeThreshold()** value, false to use the dynamically computed value.

  Notes

  The edge threshold determines which edges are considered during feature detection. The contrast threshold, as controlled by contrastThreshold(), determines whether a particular *result* is discarded or kept, based on the average contrast of all of the features in the result.

  This is an advanced parameter. In most cases, the default value should be used.

edgeThreshold

```
double edgeThreshold() const;

void edgeThreshold(double edgeThreshold);
```

- `double edgeThreshold() const;`

  Returns the user-selected edge threshold.

  Notes

  This function does not return PatMax's internally computed threshold, regardless of the value of **useEdgeThreshold()**.

- `void edgeThreshold(double edgeThreshold);`

  If **useEdgeThreshold()** is true, then the value you supply for this function is used during feature extraction. The orientation of image edges with a strength (defined as the difference in grey-level values across the edge) are randomized. If false, PatMax's dynamically computed edge threshold is used.

  The supplied value must be in the range from 0.0 through 255.0.

  The default value is 5.0.

  Parameters

  edgeThreshold The edge threshold.

  Throws

  cc_PMDefs::BadParams

  edgeThreshold is less than 0.0 or greater than 255.0.

  Notes

  The edge threshold determines which edges are considered during feature detection. The contrast threshold, as controlled by contrastThreshold(), determines whether a particular *result* is discarded or kept, based on the average contrast of all of the features in the result.

  This is an advanced parameter. In most cases, the default value should be used.

scoreUsingClutter

```
bool scoreUsingClutter() const;

void scoreUsingClutter(bool useClutter);
```

- `bool scoreUsingClutter() const;`

  Returns whether or not this **cc_PMRunParams** considers extraneous features (clutter) when computing the score of a pattern instance. If the function returns true, extraneous features are considered; if it returns false, extraneous features are ignored.

- `void scoreUsingClutter(bool useClutter);`

  Sets whether or not this **cc_PMRunParams** considers extraneous features (clutter) when computing the score of a pattern instance.

  The default is to consider clutter when computing the score of a pattern instance.

Parameters

useClutter True (the default) tells PatMax to consider extraneous features when scoring a pattern instance. False tells PatMax to ignore extraneous features when scoring a pattern instance.

numToFind

```
c_Int32 numToFind() const;
void numToFind(c_Int32 count);
```

- `c_Int32 numToFind() const;`

  Returns the number of results to search for.

- `void numToFind(c_Int32 n);`

  Sets the number of results to search for. In some cases, PatMax may return more or fewer results than you request. This function returns one result by default.

  Parameters

  n The number of results. The default is one.

  Throws

  cc_PMDefs::BadParams
  count is less than one.

zoneEnable

```
c_UInt32 zoneEnable() const;
void zoneEnable(c_UInt32 enable);
```

- `c_UInt32 zoneEnable() const;`

  Returns a **c_UInt32** that indicates which degrees of freedom are enabled in this **cc_PMRunParams**. The value returned by this function is either 0 or a value formed by ORing together one or more of the following values:

  cc_PMDefs::kAngle
  cc_PMDefs::kUniformScale
  cc_PMDefs::kXScale
  cc_PMDefs::kYScale

- `void zoneEnable(c_UInt32 enable);`

  Sets which degrees of freedom are included in the search space defined by this **cc_PMRunParams**. For each specified degree of freedom, PatMax will search for instances of the pattern with values between the low zone limit and high zone limit.

  For each degree of freedom that is *not* specified, PatMax only searches for instances of the pattern with the specified nominal value for that degree of freedom, and PatMax only returns the nominal value for the degree of freedom.

  By default, no degrees of freedom are enabled.

  Parameters

  enable — A value formed by bit-wise ORing together 0 or more of the following values:

  cc_PMDefs::kAngle
  cc_PMDefs::kUniformScale
  cc_PMDefs::kXScale
  cc_PMDefs::kYScale

  Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

  Throws

  cc_PMDefs::BadParams

  enable is neither 0 nor a value formed by bit-wise ORing together the values defined in **cc_PMDefs::DOF**.

---

nominal

```
double nominal(cc_PMDefs::DOF dof) const;

void nominal(cc_PMDefs::DOF dof, double val);
```

- `double nominal(cc_PMDefs::DOF dof) const;`

  Returns the nominal value for the specified degree of freedom.

  Parameters

  dof — The degree of freedom; *dof* must be one of the following values:

  cc_PMDefs::kAngle
  cc_PMDefs::kUniformScale
  cc_PMDefs::kXScale
  cc_PMDefs::kYScale

- `void nominal(cc_PMDefs::DOF dof, double val);`

  Sets the nominal value for the specified degree of freedom. If the specified degree of freedom has not been enabled, PatMax only searches for instances of the pattern with the specified nominal value for the specified degree of freedom, and PatMax only returns the nominal value for the degree of freedom.

  Parameters

  dof — The degree of freedom for which to set a nominal value; *dof* must be one of the following values:

  cc_PMDefs::kAngle
  cc_PMDefs::kUniformScale
  cc_PMDefs::kXScale
  cc_PMDefs::kYScale

  Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

  val — The nominal value.

  The default value for *cc_PMDefs::kAngle* is 0.0.

  The default value for *cc_PMDefs::kUniformScale*, *cc_PMDefs::kXScale*, and *cc_PMDefs::kYScale* is 1.0. These *dof*s must be assigned a value greater than 0.

  Throws

  cc_PMDefs::BadParams
  val does not meet the requirements described above.

zoneLow `double zoneLow (cc_PMDefs::DOF dof) const;`

Returns the lower zone limit for the specified degree of freedom.

Parameters

dof — The degree of freedom for which to return the lower zone limit. Must be one of the following values:

cc_PMDefs::kAngle
cc_PMDefs::kUniformScale
cc_PMDefs::kXScale
cc_PMDefs::kYScale

Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

Throws

cc_PMDefs::BadParams
If *dof* is not one of the values listed above.

zoneHigh

```
double zoneHigh (cc_PMDefs::DOF dof) const;
```

Returns the upper zone limit for the specified degree of freedom.

Parameters

dof The degree of freedom for which to return the upper zone limit. Must be one of the following values:

cc_PMDefs::kAngle
cc_PMDefs::kUniformScale
cc_PMDefs::kXScale
cc_PMDefs::kYScale

Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

Throws

cc_PMDefs::BadParams
If *dof* is not one of the values listed above.

zone

```
void zone(cc_PMDefs::DOF dof, double low, double high);
```

Sets the lower and upper zone limits for the specified degree of freedom. If the specified degree of freedom has been enabled, PatMax only returns results for instances of the trained pattern found in the search image that exhibit transformations within the specified zones of the specified degrees of freedom.

Parameters

dof The degree of freedom for which to set the zone limits; *dof* must be one of the following values:

cc_PMDefs::kAngle
cc_PMDefs::kUniformScale
cc_PMDefs::kXScale
cc_PMDefs::kYScale

Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

low The lower zone limit. Default values are as follows

DOF	Default
cc_PMDefs::kAngle	-45°
cc_PMDefs::kUniformScale	0.8
cc_PMDefs::kXScale	0.8
cc_PMDefs::kYScale	0.8

high The upper zone limit. Default values are as follows:

DOF	Default
cc_PMDefs::kAngle	+45°
cc_PMDefs::kUniformScale	1.2
cc_PMDefs::kXScale	1.2
cc_PMDefs::kYScale	1.2

Notes

For the uniform scale DOF, the scales are equal in both the x and y dimensions, while the x-scale and y-scale DOFs scale only in the given dimension. For all of these DOFs, the lower zone limit must be less than or equal to the upper zone limit.

The scale DOFs are specified as a multiplier of the trained pattern size. For example, a uniform scale of 1.25 implies that the runtime (client coordinate) size of the pattern is 25% larger in both dimensions than the trained pattern.

The angle zone is always searched from the low to high. If the lower zone limit is greater than the upper zone limit, the range of angles that is searched is inverted. For example, setting **zoneLow()** to 0 and **zoneHigh()** to 180 searches one half of the 360° angle range; setting **zoneHigh()** to 0 and **zoneLow()** to 180 searches the other half.

Throws

cc_PMDefs::BadParams

dof is not one of the values listed above,
or *dof* is not equal to *kAngle* and *low* is greater than *high,*
or *dof* is not equal to *kAngle* and *low* is less than 0.

xyOverlap

```
double xyOverlap() const;
void xyOverlap(double overlapThresh);
```

- `double xyOverlap() const;`

  Returns the percentage of area overlap that this **cc_PMRunParams** requires before treating overlapping pattern instances as a single pattern. Two results are considered overlapping if the value returned by this function is greater than **overlapThresh()**.

  Notes

  The overlap value is approximately the ratio of the overlap area to the total area of the pattern. Meaningful values range from 0.0 to 1.0. The default overlap is 0.8

- `void xyOverlap(double overlapThresh);`

  Sets the percentage of area overlap that this **cc_PMRunParams** requires before treating overlapping pattern instances as a single pattern.

  Parameters

  overlapThresh The percentage of overlap from 0.0 to 1.0.

  A value of 0.0 means that all pattern instances with any non-zero amount of intersection are considered overlapping, and are treated as a single pattern.

  A value of 1.0 means that no pattern instances that intersect each other are considered overlapping (overlap is disabled). Each pattern instance is treated separately.

  The default overlap threshold is 0.8.

  Throws

  cc_PMDefs::BadParams

  overlapThresh is outside of the range of 0.0 through 1.0.

zoneOverlap

```
double zoneOverlap(cc_PMDefs::DOF dof) const;
void zoneOverlap(cc_PMDefs::DOF dof, double overlapThresh);
```

- `double zoneOverlap(cc_PMDefs::DOF dof) const;`

  Returns the overlap threshold for the specified degree of freedom. When two pattern instances lie within the specified overlaps for all enabled degrees of freedom and within the area overlap threshold, PatMax discards weaker pattern instances.

Parameters

dof The degree of freedom for which to return the zone overlap; *dof* must be one of the following values:

cc_PMDefs::kAngle
cc_PMDefs::kUniformScale
cc_PMDefs::kXScale
cc_PMDefs::kYScale

Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

- `void zoneOverlap(cc_PMDefs::DOF dof, double overlapThresh);`

Sets the overlap threshold for the specified degree of freedom. When two pattern instances lie within the specified overlaps for all enabled degrees of freedom and within the area overlap threshold, PatMax discards weaker pattern instances.

Parameters

dof The degree of freedom for which to set the zone overlap; *dof* must be one of the following values:

cc_PMDefs::kAngle
cc_PMDefs::kUniformScale
cc_PMDefs::kXScale
cc_PMDefs::kYScale

Scope the enumerations for the degree of freedom using the appropriate name space class such as **ccPMParamsDefs**.

overlapThresh The overlap threshold.

The units in which the overlap threshold is expressed vary depending on the degree of freedom you specify.

For the angle degree of freedom, the overlap threshold is expressed as the absolute difference between the angles of the instances.

For any scale degree of freedom, the overlap threshold is expressed as the ratio of the scale of the instance with a larger scale to the scale of the instance with the smaller scale.

For example, if you specify 1.2 for *overlapThresh* and two instances had scales of 1.25 and 2.50, they would not meet the overlap threshold because the ratio of 2.5 to 1.25 (2.0) is greater than the threshold value (1.2).

The default overlap threshold for the angle DOF is 360.0. The default overlap threshold for any of the scale DOFs is 1.4.

Throws

cc_PMDefs::BadParams

overlap is invalid for the specified degree of freedom.

timeOut

```
double timeOut() const;
void timeOut(double timeInSeconds);
```

- `double timeOut() const;`

Returns the timeout value for this **cc_PMRunParams**.

- `void timeOut(double timeInSeconds);`

Sets the timeout value for this **cc_PMRunParams**. The timeout value limits how long **ccPMAlignPattern::run()** or **ccPMInspectPattern::run()** execute before throwing a *ccTimeout::Expired* error. If this error is thrown, the tool does not return any valid results.

Parameters

timeInSeconds The timeout value in seconds. The default timeout is *HUGE_VAL*, which allows infinite execution time.

Notes

The timeout value is intended as an approximate upper limit. Due to internal latencies, the actual execution time can exceed the requested maximum by up to 0.02 seconds (on a 200 Mhz Pentium I). Typical latencies are much lower. Latencies will also be lower on faster CPUs.

The timeout is implemented using the **ccTimeout** class defined in *<ch_cvl/attent.h>*. User-created **ccTimeout** objects can also terminate the **run()** function.

Throws

cc_PMDefs::BadParams

timeInSeconds is less than 0.

flexRunParams

```
ccPMFlexRunParams flexRunParams() const;
void flexRunParams(const ccPMFlexRunParams &params);
```

Specifies the PatFlex run-time parameters used with the PatFlex algorithm.

Notes

This function has no effect unless the PatMax algorithm is *kPatflex*.

This function has no effect on normal PatMax patterns.

- `ccPMFlexRunParams flexRunParams() const;`

  Returns the PatFlex run-time parameters object.

- `void flexRunParams(const ccPMFlexRunParams &params);`

  Sets a new PatFlex run-time parameters object.

Parameters

params	The new PatFlex run-time parameters object.

outsideRegionThreshold

```
double outsideRegionTheshold() const;

void outsideRegionThreshold(double
                                outsideRegionThreshold);
```

Gets and sets the outside region threshold used by PatMax and PatQuick. Setting *outsideRegionThreshold* greater than 0 enables patterns that extend outside the search region to be found by PatMax without penalizing the score. That is, a pattern that extends partially outside the search region can still achieve a score of 1.0.

The *outsideRegionThreshold* parameter specifies what proportion of the pattern's features can be outside the search region without penalizing the score. If the proportion of pattern features lying outside the search region is greater than *outsideRegionThreshold,* then the pattern will not be found.

This functionality allows users to train an entire field of view, and then search for that trained field of view on a runtime image. By setting *outsideRegionThreshold*, PatMax can accommodate translation between the trained field of view and the runtime scene.

Often, using a larger training region improves discrimination between candidate matches. This functionality allows users to train larger patterns without worrying that the entire pattern must appear in the runtime image.

- `double outsideRegionTheshold() const;`

  Returns the current *outsideRegionThreshold* setting.

- `void outsideRegionThreshold(double outsideRegionThreshold);`

  Sets the outside region threshold for this **cc_PMRunParams**. The default value is 0.0.

  Parameters

  outsideRegionThreshold

  A value between 0.0 and 0.999, inclusive, representing the proportion of pattern features that can fall outside the search region without penalizing the score amount. For example, *outsideRegionThreshold* = 0.20 specifies that 20% of the pattern's features can fall outside the search region.

  Notes

  Setting an outside region threshold value significantly increases the running time for PatMax and PatQuick.

  Once you have made a test run of PatMax with *outsideRegionThreshold* > 0, you can use the value reported by **cc_PMResult::outsideRegionFeatureProportion()** to hone in on a more accurate *outsideRegionThreshold* setting. However, in some cases, for the pattern to be found, *outsideRegionThreshold* must be set 0.05 to 0.10 larger than the measured result reported by **outsideRegionFeatureProportion()**. For this reason, Cognex recommends setting *outsideRegionThreshold* so that it exceeds the maximum expected outside region proportion by 0.10. That is, set *outsideRegionThreshold* = (maximum expected outside region proportion) + 0.10.

  The *outsideRegionThreshold* parameter trades off speed for robustness. PatMax runs faster when a smaller *outsideRegionThreshold* value is used, but PatMax may miss the pattern if the threshold is too small.

  Throws

  BadParams

  If *outsideRegionThreshold* is less than 0.0 or greater than 0.999.

  Notes

  ccPMAlignPattern::run() throws *NotImplemented* if you specify both an algorithm of *kPatflex* or *kPatpersp* and *outsideRegionThreshold* > 0.0.

cc_PMRunParams

cc_PMStageResult

```
#include <ch_cvl/pmpbase.h>

class cc_PMStageResult
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

A class that contains a single PatMax search result from one stage of alignment.

Constructors/Destructors

cc_PMStageResult

```
cc_PMStageResult();
```

Constructs an empty PatMax Stage Result object.

Public Member Functions

location

```
cc2Vect location() const;
```

Returns the location of the pattern origin (in the client coordinate system of the runtime image).

pose

```
const cc2Xform& pose() const;
```

Returns a transform from the training time client coords (translated to the pattern origin) to runtime client coords. Map **cc2Vect(0,0)** through this to get the location of the pattern origin in runtime client coords.

angle

```
ccDegree angle() const;
```

Returns the angle of the located pattern (in the client coordinate system of the runtime image).

xScale

```
double xScale() const;
```

Returns the x-scale of the located pattern.

yScale `double yScale() const;`

Returns the y-scale of the located pattern.

score `double score() const;`

Returns the score, a number between 0.0 and 1.0.

fitError `double fitError() const;`

A PatMax quality measure in the range 0.0 (best) to infinity (worst). While there is no upper limit, *fitError* is typically less than 10. A -1 is returned if the measure is not available.

coverage `double coverage() const;`

A PatMax quality measure in the range 0.0 (worst) to 1.0 (best). A -1 is returned if the measure is not available.

clutter `double clutter () const;`

A PatMax quality measure in the range 0.0 (best) to infinity (worst). While there is no upper limit, *clutter* is typically less than 10. A -1 is returned if the measure is not available.

matchRegion `ccAffineRectangle matchRegion() const;`

Returns an affine rectangle which describes the location of the matched pattern in the runtime image's client coordinates.

imageRegion `ccPelRect imageRegion() const;`

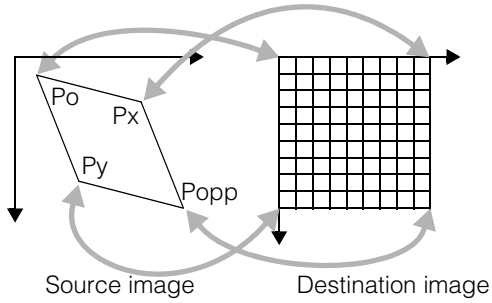
Returns the smallest rectangle in runtime image's image coordinates which completely contains the matched model.

cfAffineTransformImage()

```
#include <ch_cvl/afftrans.h>

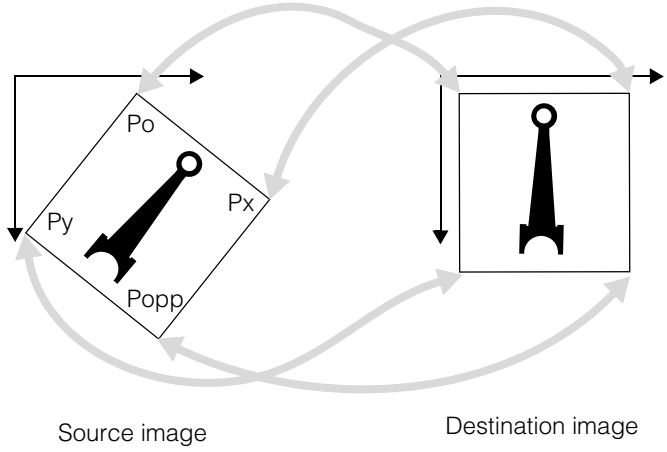
cfAffineTransformImage();
```

Global function to perform an affine sampling transformation on an input image. An affine rectangle in the source image is sampled according to a set of affine sampling parameters and the sampled points are transformed to a destination image through a destination client coordinate transform that the routine creates. The transformed affine rectangle maps the destination image as shown below for the general case:



Note that the outermost corners of the corner pixels (not the pixel center) in the destination image map to the corners of the affine rectangle in the input image. (Also see the **ccAffineRectangle** and **ccAffineSamplingParams** reference pages).

A common use of this function is for image rotation or for small corrections to scale. For example the following transformation corrects for rotation:



The function has three overloads which perform the following:

a. The first overload maps a source image to a destination image. The destination image can be bound or unbound. The handling of these bound and unbound cases is described below. If the affine rectangle clips the source image (falls outside the source image), the function throws and error.

b. The second overload is similar to (a) above except that it allows clipping. You supply it a *weights* image which it fills in to mark transformed pixels when clipping occurs.

c. The third overload takes only a source image and then returns a transformed destination image that it creates. Like (a) above, if the affine rectangle clips the source image, the function throws and error.

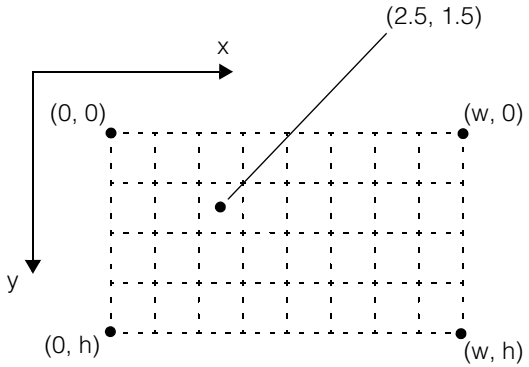
Unbound destination images

If the destination image you provide is unbound, the resulting destination *clientFromImage* transform will map image coordinates to source client coordinates as follows:

Destination image coordinates		Source client coordinates
(0, 0)	---- Maps to ---->	Po
(w, 0)	---- Maps to ---->	Px
(0, h)	---- Maps to ---->	Py
(w, h)	---- Maps to ---->	Popp

Where w and h are the width and height of the destination image as specified in the sampling parameters. Po, Px, Py, and Popp are defined in the **ccAffineRectangle** reference page and also shown in the diagrams on the previous page.

It is important to realize that the image coordinate points (0, 0), (h, w), and so on, represent the outer corners of the image pixels and not the pixel centers. See the following destination image diagram:



Destination image

When the destination pel buffer is unbound it contains no useful information about the destination image pixels. For unbound destination images the routine creates a new bound **ccPelBuffer** of the appropriate size for the transformation, and copies it into the destination pel buffer provided. All offsets are set to zero.

Bound destination images

If the destination is bound, samples are written to the destination as if the following steps had taken place:

1. A temporary destination image, TMP, was created using the rules stated above for unbound destination images.
2. The function **cfPelCopy(**TMP, *destination***)** was used to transfer pixels from TMP to the bound destination. This operation will only copy pixels within the Greatest Common Rectangle (GCR) of TMP and destination. (That is, when the TMP pel buffer is superimposed onto the destination pel buffer, only pixel coordinates common to both pel buffers are copied).

The actual implementation may be different, but will have the same result. You should be certain that you understand the steps above to ensure correct operation. In some cases it may be necessary to shift the destination image offset to achieve the desired result. This function does not re-size, re-allocate, or alter the offsets of a bound destination.

A bound destination image must not share any of the source image pixels that will be sampled.

cfAffineTransformImage

```
template<class T>
void cfAffineTransformImage(
  const ccPelBuffer_const<T>& srcImage,
  ccPelBuffer<T>& dstImage,
  const ccAffineSamplingParams& params);

template<class T>
void cfAffineTransformImage(
  const ccPelBuffer_const<T>& srcImage,
  ccPelBuffer<T>& dstImage,
  ccPelBuffer<T>& weights,
  const ccAffineSamplingParams& params,
  bool& clipped);

template<class T>
ccPelBuffer<T> cfAffineTransformImage(
  const ccPelBuffer_const<T>& srcImage,
  const ccAffineSamplingParams& params);
```

-
```
template<class T>
void cfAffineTransformImage(
  const ccPelBuffer_const<T>& srcImage,
  ccPelBuffer<T>& dstImage,
  const ccAffineSamplingParams& params);
```

Samples the *srcImage* as specified by *params*, and places the results in *dstImage*. The client coordinate transform of the destination image is created by the routine and it transforms points in the destination image to points in the source image.

Parameters

T — A template parameter specifying the type of the source and destination images. *T* must be **c_UInt8** or **c_UInt16**.

srcImage — The image to sample

dstImage — The image in which to place the sampled image

params — The sampling parameters to use

Throws

ccAffTransDefs::UnboundWindow
If *srcImage* is unbound.

ccAffTransDefs::Clipped
If clipping occurs in the source image. For example, if any sampled point is outside the source image.

ccPelFunc::Overlap
: *dstImage* is bound and shares pixels with *srcImage*.

ccAffineSamplingParams::NotImplemented
: *params.interpolation* is *ccAffineSamplingParams::eBilinearApprox* or *ccAffineSamplingParams::eHighPrecision* and *srcImage.rowUpdate()* or *srcImage.height()* is greater than or equal to 32768.

- ```
  template<class T>
  void cfAffineTransformImage(
    const ccPelBuffer_const<T>& srcImage,
    ccPelBuffer<T>& dstImage,
    ccPelBuffer<T>& weights,
    const ccAffineSamplingParams& params,
    bool& clipped);
  ```

  Functions identically to the overload above, except it allows source image clipping.

  If the input image is clipped by the supplied **ccAffineSamplingParams**, a *weights* image is computed and placed in the *weights* pel buffer you provide, and the *clipped* flag is set true. The *weights* image is filled with mask values between 0 and 255 to indicate which corresponding pixels in the destination image represent valid transformations. The mask values have the following meaning:

  - A mask value of 255 indicates that the corresponding destination pixel value was derived entirely from pixels within the source image (no clipping).
  - A mask value of 0 indicates that the corresponding destination pixel value was derived entirely from pixels outside the source image (total clipping). Pixels outside the source image are assumed to be 0.
  - Mask values from 1 through 254 correspond to boundary samples where the sample was derived from both pixels in the source image and clipped pixels. When the sampling algorithm includes clipped pixels in its calculations, it uses the value 0 for these pixels.

  Parameters

T	A template parameter specifying the type of the source and destination images. *T* must be **c_UInt8** or **c_UInt16**.
srcImage	The image to sample
dstImage	The image in which to place the sampled image
weights	The weights image as described above
params	The sampling parameters to use
  ```

clipped If *true*, clipping occurred. If *false*, no clipping occurred and *weights* was not modified.

ccAffineSamplingParams::NotImplemented
params.interpolation is *ccAffineSamplingParams::eBilinearApprox* or *ccAffineSamplingParams::eHighPrecision* and *srcImage.rowUpdate()* or *srcImage.height()* is greater than or equal to 32768.

Throws

ccAffTransDefs::UnboundWindow
If *srcImage* is unbound.

ccPelFun::Overlap
dstImage is bound and shares pixels with *srcImage*.

- ```
  template<class T>
  ccPelBuffer<T> cfAffineTransformImage(
    const ccPelBuffer_const<T>& srcImage,
    const ccAffineSamplingParams& params);
  ```

Samples the *srcImage* as specified by *params*, and returns the results in a newly constructed destination image. This is essentially the same as the first overload (unbound) except that the created destination pel buffer is returned to the caller.

Parameters

T A template parameter specifying the type of the source and destination images. *T* must be **c_UInt8** or **c_UInt16**.

srcImage The image to sample

params The sampling parameters to use

Throws

ccAffTransDefs::UnboundWindow
If *srcImage* is unbound.

ccAffTransDefs::Clipped
If clipping occurs in the source image. For example, if any sampled point is outside the source image.

ccAffineSamplingParams::NotImplemented
params.interpolation is *ccAffineSamplingParams::eBilinearApprox* or
  ```

ccAffineSamplingParams::eHighPrecision and *srcImage.rowUpdate()* or *srcImage.height()* is greater than or equal to 32768.

cfAffineTransformImage()

cfAutoSelect()

```
#include <ch_cvl/atslptmx.h> // If using PatMax
#include <ch_cvl/atslcnls.h> // If using CNLSearch

cfAutoSelect();
```

A templated global function that runs the Auto-select tool on an input image.

Note If running Auto-select with PatMax, you must include the *<ch_cvl/atslptmx.h>* header file. If running Auto-select with CNLSearch, you must include the *<ch_cvl/atslcnls.h>* header file. Omitting the proper header file for the vision tool you are running will cause linker errors.

The **cfAutoSelect()** function is instantiated for the following types:

- **ccCnlSearchRunParams**
- **ccPMAlignRunParams**

To invoke the **cfAutoSelect()** function, allocate and initialize an instance of one of the supported types for *P*, then supply it to **cfAutoSelect()**. The function will locate and return an appropriate region for use as a model with the specified pattern-location tool. If this function does not return any model candidates, try selecting a different sampling rate and specifying a different model size.

Inputs to all overloads of the **cfAutoSelect()** function are an input image, a **ccAutoSelectParams** object specifying the auto-selection parameters, an object specifying the runtime parameters of the pattern-location tool (see above two bullets), and a container to hold the **ccAutoSelectResult** instances that the Auto-select tool returns. Optional inputs are an image/position mask and a vector of specific locations to query, both of which are valid only when running the tool in enhanced mode.

cfAutoSelect

```
template<class P>
void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
  const ccAutoSelectParams& params,
  const P& modelRunParams,
  cmStd vector<ccAutoSelectResult>& results);

template<class P>
void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
  const cmStd vector<ccIPair>& at,
  const ccAutoSelectParams& params,
  const P& modelRunParams,
  cmStd vector<ccAutoSelectResult>& results);

template<class P>
void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
```

```
  const ccAutoSelectParams& params,
  const P& modelRunParams,
  cmStd vector<ccAutoSelectResult>& results);

template<class P>
void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const cmStd vector<ccIPair>& at,
  const ccAutoSelectParams& params,
  const P& modelRunParams,
  cmStd vector<ccAutoSelectResult>& results);
```

- ```
  template<class P>
  void cfAutoSelect(
    const ccPelBuffer_const<c_UInt8>& image,
    const ccAutoSelectParams& params,
    const P& modelRunParams,
    cmStd vector<ccAutoSelectResult>& results);
  ```

  Runs a search operation of the Auto-select tool without any image/position mask.

  Parameters

  image The image in which to locate an appropriate model.

  params **ccAutoSelectParams** containing the parameters for the Auto-select tool.

  modelRunParams
  A run-time parameters object for the pattern-location tool for which you want to locate a model.

  results A reference to a vector of **ccAutoSelectResult** into which the results will be placed. Any existing contents of *results* are cleared.

  Throws

  ccAutoSelectDefs::BadModel
  The model size specified in *params* is too small or too large to allow reliable extraction of data.

  ccAutoSelectDefs::BadSample
  The sub-sampling factor specified in *params* is less than or equal to 0.

  ccAutoSelectDefs::BadWeights
  Any one of the score combination weights specified in *params* is less than 0, or all of the weights are 0.
  ```

ccAutoSelectDefs::BadResults
The maximum number of results specified in *params* is less than or equal to 0.

ccAutoSelectDefs::NotImplemented
The score combination method or direction provided in *params* is not supported.

- ```
  template<class P>
  void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
    const cmStd vector<ccIPair>& at,
    const ccAutoSelectParams& params,
    const P& modelRunParams,
    cmStd vector<ccAutoSelectResult>& results);
  ```

  Runs a query operation of the Auto-select tool without any image/position mask.

  Parameters

  image The image in which to locate an appropriate model.

  at Vector of locations at which to run the query.

  params **ccAutoSelectParams** containing the parameters for the Auto-select tool.

  modelRunParams
  A run-time parameters object for the pattern-location tool for which you want to locate a model.

  results A reference to a vector of **ccAutoSelectResult** into which the results will be placed. Any existing contents of *results* are cleared.

  Throws

  ccAutoSelectDefs::BadModel
  The model size specified in *params* is too small or too large to allow reliable extraction of data.

  ccAutoSelectDefs::BadSample
  The sub-sampling factor specified in *params* is less than or equal to 0.

  ccAutoSelectDefs::BadWeights
  Any one of the score combination weights specified in *params* is less than 0, or all of the weights are 0.
  ```

ccAutoSelectDefs::BadResults
The maximum number of results specified in *params* is less than or equal to 0.

ccPel::BadCoord
Any of the query locations specified for a query operation results in a window location that is either too close to the edge of the image or extends beyond it.

ccAutoSelectDefs::NotEnhancedMode
A query operation is being invoked, but the enhanced mode is not enabled (that is, **ccAutoSelectParams::isEnhancedMode()** returns false).

ccAutoSelectDefs::NotImplemented
The score combination method or direction provided in *params* is not supported.

- ```
  template<class P>
  void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccAutoSelectParams& params,
    const P& modelRunParams,
    cmStd vector<ccAutoSelectResult>& results);
  ```

  Runs a search operation of the Auto-select tool with an image/position mask.

  Parameters

  image The image in which to locate an appropriate model.

  mask Image/position mask used to filter out certain areas of the input image from the analysis.

  params **ccAutoSelectParams** containing the parameters for the Auto-select tool.

  modelRunParams
  A run-time parameters object for the pattern-location tool for which you want to locate a model.

  results A reference to a vector of **ccAutoSelectResult** into which the results will be placed. Any existing contents of *results* are cleared.

  Throws

  ccAutoSelectDefs::BadModel
  The model size specified in *params* is too small or too large to allow reliable extraction of data.
  ```

ccAutoSelectDefs::BadSample
The sub-sampling factor specified in *params* is less than or equal to 0.

ccAutoSelectDefs::BadWeights
Any one of the score combination weights specified in *params* is less than 0, or all of the weights are 0.

ccAutoSelectDefs::BadResults
The maximum number of results specified in *params* is less than or equal to 0.

ccAutoSelectDefs::NotEnhancedMode
An image/position mask is being provided and the enhanced mode is not enabled.

ccAutoSelectDefs::BadMask
The window mask supplied in *params* is a bound pel buffer of any size other than the model size specified in *params*, or the size of *mask* differs from that of the search image.

ccAutoSelectDefs::NotImplemented
The score combination method or direction provided in *params* is not supported.

-
```
template<class P>
void cfAutoSelect(const ccPelBuffer_const<c_UInt8>& image,
  const ccPelBuffer_const<c_UInt8>& mask,
  const cmStd vector<ccIPair>& at,
  const ccAutoSelectParams& params,
  const P& modelRunParams,
  cmStd vector<ccAutoSelectResult>& results);
```

Runs a query operation of the Auto-select tool with an image/position mask.

Parameters

image The image in which to locate an appropriate model.

mask Image/position mask used to filter out certain areas of the input image from the analysis.

at Vector of locations at which to run the query operation.

params **ccAutoSelectParams** containing the parameters for the Auto-select tool.

modelRunParams
A run-time parameters object for the pattern-location tool for which you want to locate a model.

results A reference to a vector of **ccAutoSelectResult** into which the results will be placed. Any existing contents of *results* are cleared.

Throws

ccAutoSelectDefs::BadModel
The model size specified in *params* is too small or too large to allow reliable extraction of data.

ccAutoSelectDefs::BadSample
The sub-sampling factor specified in *params* is less than or equal to 0.

ccAutoSelectDefs::BadWeights
Any one of the score combination weights specified in *params* is less than 0, or all of the weights are 0.

ccAutoSelectDefs::BadResults
The maximum number of results specified in *params* is less than or equal to 0.

ccAutoSelectDefs::NotEnhancedMode
An image/position mask is being provided, but the enhanced mode is not enabled.
A query operation is being invoked, but the enhanced mode is not enabled (that is, **ccAutoSelectParams::isEnhancedMode()** returns false).

ccPel::BadCoord
Any of the query locations specified for a query operation results in a window location that is either too close to the edge of the image or extends beyond it.

ccAutoSelectDefs::BadMask
The window mask supplied in *params* is a bound pel buffer of any size other than the model size specified in *params*, or the size of *mask* differs from that of the search image.

ccAutoSelectDefs::NotImplemented
The score combination method or direction provided in *params* is not supported.

cfAutoTrigger()

```
#include <ch_cvl/trigmodl.h>

cfAutoTrigger();
```

Global function to retrieve an auto trigger model object reference.

cfAutoTrigger

```
const ccTriggerModel& cfAutoTrigger();
```

When you specify **cfAutoTrigger()** as the trigger model, the application automatically starts an acquisition when there is a transition on the trigger input line. The auto trigger model is sometimes called a “hardware trigger.” There is a fixed association between camera ports and trigger input lines.

If **ccTriggerProp::triggerEnable()** is set to false, transitions on the input line are ignored. For acquisitions to proceed, **ccTriggerProp::triggerEnable()** must be set to true.

Notes

If you change property values while **ccTriggerProp::triggerEnable()** is set to true, the changes will take place after an unspecified number of acquisitions. To force changes in property values to take effect with the next acquisition, set **triggerEnable()** to false, change the properties, then set **triggerEnable()** to true. Alternatively, you may want to consider using the semi trigger model.

If you call **ccAcqFifo::start()** when the auto trigger model is selected, your application throws *ccAcqFifo::StartNotAllowed*.

The default behavior is for a low-to-high transition on the trigger input line to trigger an acquisition. You can change this behavior to cause a high-to-low transition to trigger an acquisition by setting **ccTriggerProp::triggerLowToHigh()** to false.

cfAutoTrigger()

cfBlobAnalysis()

```
#include <ch_cvl/blobtool.h>

cfBlobAnalysis();
```

Global function to perform blob analysis on an input image.

cfBlobAnalysis

```
template<class T>
void cfBlobAnalysis(const T& image,
  const ccBlobParams& params, ccBlobResults& results,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);
```

Performs a blob analysis of the specified image using the specified parameters.

This is a templated function that supports both standard **ccPelBuffer<c_UInt8>** images and **ccRLEBuffer** images.

Parameters

image — The image to analyze. You can specify either a **ccPelBuffer<c_UInt8>** image or a **ccRLEBuffer** image.

params — A reference to a **ccBlobParams** that specifies the parameters to use.

results — A reference to a **ccBlobResults**. The results of the blob analysis are placed in *results*.

dObj — An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags — The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccBlobDefs::BadParams
The analysis is equal to *ccBlobSceneDescription::eWholeImageGreyScale* and **params.connectivityCleanup()** is not equal to *ccBlobSceneDescription::eNone*.
The image is unbound or degenerate.

cfBlobAnalysis()

cfBoundaryTracker()

```
#include <ch_cvl/bndtrckr.h>

cfBoundaryTracker();
```

Global function to apply the Boundary Tracker tool to an input image. For more information, see the chapter *Boundary Tracker Tool* in the *CVL Vision Tools Guide*.

cfBoundaryTracker

```
void cfBoundaryTracker(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccBoundaryTrackerRunParams& params,
  ccBoundaryTrackerResult& result);
```

Optionally locates, then tracks the boundary of an object in the supplied image.

Parameters

image	The image in which to track an object boundary. *image* must exhibit a bimodal distribution of pixel values between object and background pixels.
params	A reference to a **ccBoundaryTrackerRunParams** that specifies the searching and tracking parameters.
result	A reference to a **ccBoundaryTrackerResult** into which the results are placed.

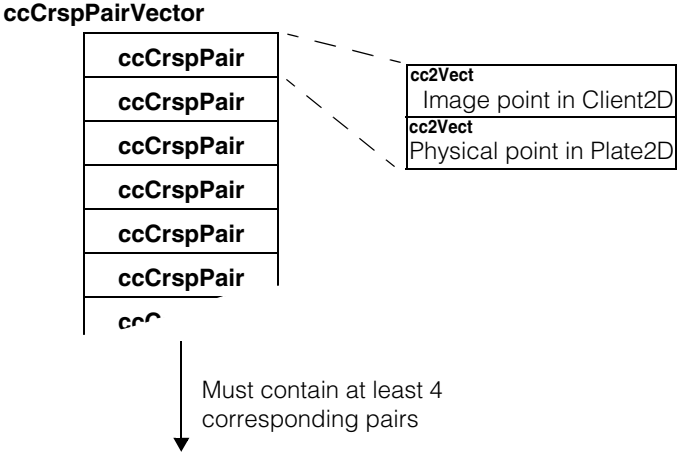
cfBoundaryTracker()

cfCalib2VertexFeatureExtract()

```
#include <ch_cvl/calibftr.h>

cfCalib2VertexFeatureExtract();
```

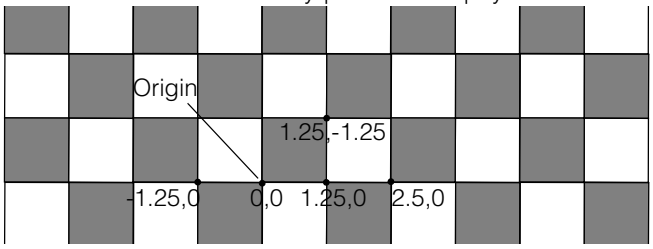
This global function resizes and initializes the supplied *client2DPlate2DPairs* vector with extracted vertex points and their corresponding coordinates in the Plate2D coordinate system. The first **cc2Vect** in each **ccCrspPair** represents the Client2D location of the vertex, while the second **cc2Vect** represents its Plate2D position. See the following diagram.



The *image points* in the diagram above are reported in Client2D units as transformed by the source image client coordinate transform. This differs from the grid-of dots calibration where source image locations are described in image coordinates. If you are using the default client coordinate transform of the source image pel buffer (the identity transform), then client units are the same as pixels and *image points* and pixel coordinates are the same.

The *physical points* in the diagram above report the physical location of the found features in Plate2D. You can define Plate2D in several ways. See **ccCalib2VertexFeatureDefs::CorrespondMethod**. The *physical point* location is a

multiple of the checkerboard pitch which you specify in **ccCalib2VertexFeatureParams**. The following diagram shows an example of physical points for a checkerboard with an x- and y-pitch of 1.25 physical units.



The **ccCrspPair** point pairs in the **ccCrspPairVector** are ordered in a raster fashion using their physical position coordinates.

Requires that the size of checkers in the image be at least 15x15 pixels. If Data Matrix codes are used as calibration fiducials, the Data Matrix codes should have a minimum resolution of 4 ppm.

cfCalib2VertexFeatureExtract

```
void cfCalib2VertexFeatureExtract(
  const ccPelBuffer_const<c_UInt8>& srcImg,
  const ccCalib2VertexFeatureParams& params,
  ccCrspPairVector& client2DPlate2DPairs,
  ccDiagObject* diagObj=0,
  c_UInt32 diagFlags=0
);

void cfCalib2VertexFeatureExtract(
  const ccPelBuffer_const<c_UInt8>& srcImg,
  const ccCalib2VertexFeatureParams& params,
  ccCrspPairWeightedVector& client2DPlate2DPairs,
  ccDiagObject* diagObj=0,
  c_UInt32 diagFlags=0
);

void cfCalib2VertexFeatureExtract(
  const ccPelBuffer_const<c_UInt8>& srcImg,
  const ccCalib2VertexFeatureParams& params,
  ccCalib2VertexFeatureResult& result,
  ccDiagObject* diagObj=0,
  c_UInt32 diagFlags=0
);
```

-
```
void cfCalib2VertexFeatureExtract(
  const ccPelBuffer_const<c_UInt8>& srcImg,
  const ccCalib2VertexFeatureParams& params,
  ccCrspPairVector& client2DPlate2DPairs,
```

```
  ccDiagObject* diagObj=0,
  c_UInt32 diagFlags=0
);
```

Parameters

srcImg The image containing the calibration plate. The size of checkerboard squares in the image must be at least 15x15 pixels. If Data Matrix codes are used as calibration fiducials, the Data Matrix codes should have a minimum resolution of 4 ppm.

params The run-time parameters for this function.

client2DPlate2DPairs
The results vector where the function places the results.

diagObj An optional **ccDiagObject**. If you supply a value for diagObj, then the tool will record diagnostic information in the supplied object.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

If **params.labelMode()** is *eUseDataMatrixWithGridPitch*, then each Data Matrix code on the calibration plate must encode the grid pitch in addition to the grid position. The value of grid pitch encoded in the Data Matrix code is used by the tool, **params.physicalGridPitch()** is ignored.

If **params.labelMode()** is *eUseDataMatrix*, then each Data Matrix code must encode the grid position and may optionally encode the grid pitch. However, the value of grid pitch encoded in the Data Matrix codes will not be used by the tool. The tool will use **params.physicalGridPitch()**.

Throws

ccPel::UnboundWindow
If *srcImg* is not bound to any root image.

ccCalib2VertexFeatureDefs::BadParam
If client transform of *srcImg* is not linear.

ccCalib2VertexFeatureDefs::BadParam
: If **params.algorithm()** is *eExhaustiveMultiRegion* and **params.labelMode()** is neither *eDataMatrix* nor *eDataMatrixWithGridPitch*

ccCalib2VertexFeatureDefs::FailedCorrespondence
: If consistent labeling of the vertices could not be carried out. This throw may occur due to out-of-focus images or images with severe light gradient or very small checker size in images, and so on.

ccCalib2VertexFeatureDefs::FiducialNotFound
: If running in *ccCalib2VertexFeatureDefs::eUseFiducial* mode and the expected fiducial feature is not found.

ccCalib2VertexFeatureDefs::DataMatrixNotFound
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix could be found in *srcImg*.

ccCalib2VertexFeatureDefs::DataMatrixDecodeError
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix code that was successfully found in *srcImg* could be successfully decoded.

ccCalib2VertexFeatureDefs::DataMatrixParseError
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix code in *srcImg* that was successfully decoded could be successfully parsed.

ccCalib2VertexFeatureDefs::DataMatrixParseError
: If **labelMode()** is *eUseDataMatrixWithGridPitch* and either component of the grid pitch extracted from any Data Matrix code in <*srcImg*> is less than or equal to zero.

- ```
  void cfCalib2VertexFeatureExtract(
    const ccPelBuffer_const<c_UInt8>& srcImg,
    const ccCalib2VertexFeatureParams& params,
    ccCrspPairWeightedVector& client2DPlate2DPairs,
  ```
  ```

```
  ccDiagObject* diagObj=0,
  c_UInt32 diagFlags=0
);
```

Parameters

srcImg The image containing the calibration plate. The size of checkerboard squares in the image must be at least 15x15 pixels. If Data Matrix codes are used as calibration fiducials, the Data Matrix codes should have a minimum resolution of 4 ppm.

params The run-time parameters for this function.

client2DPlate2DPairs
The weighted results vector where the function places the results.

diagObj An optional **ccDiagObject**. If you supply a value for diagObj, then the tool will record diagnostic information in the supplied object.

diagFlags Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

If **params.labelMode()** is *eUseDataMatrixWithGridPitch*, then each Data Matrix code on the calibration plate must encode the grid pitch in addition to the grid position. The value of grid pitch encoded in the Data Matrix code is used by the tool, **params.physicalGridPitch()** is ignored.

If **params.labelMode()** is *eUseDataMatrix*, then each Data Matrix code must encode the grid position and may optionally encode the grid pitch. However, the value of grid pitch encoded in the Data Matrix codes will not be used by the tool. The tool will use **params.physicalGridPitch()**.

Throws

ccPel::UnboundWindow
If *srcImg* is not bound to any root image.

ccCalib2VertexFeatureDefs::BadParam
If client transform of *srcImg* is not linear.

ccCalib2VertexFeatureDefs::BadParam
: If **params.algorithm()** is *eExhaustiveMultiRegion* and **params.labelMode()** is neither *eDataMatrix* nor *eDataMatrixWithGridPitch*

ccCalib2VertexFeatureDefs::FailedCorrespondence
: If consistent labeling of the vertices could not be carried out. This throw may occur due to out-of-focus images or images with severe light gradient or very small checker size in images, and so on.

ccCalib2VertexFeatureDefs::FiducialNotFound
: If running in *ccCalib2VertexFeatureDefs::eUseFiducial* mode and the expected fiducial feature is not found.

ccCalib2VertexFeatureDefs::DataMatrixNotFound
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix could be found in *srcImg*.

ccCalib2VertexFeatureDefs::DataMatrixDecodeError
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix code that was successfully found in *srcImg* could be successfully decoded.

ccCalib2VertexFeatureDefs::DataMatrixParseError
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix code in *srcImg* that was successfully decoded could be successfully parsed.

ccCalib2VertexFeatureDefs::DataMatrixParseError
: If **labelMode()** is *eUseDataMatrixWithGridPitch* and either component of the grid pitch extracted from any Data Matrix code in <*srcImg*> is less than or equal to zero.

- ```
  void cfCalib2VertexFeatureExtract(
    const ccPelBuffer_const<c_UInt8>& srcImg,
    const ccCalib2VertexFeatureParams& params,
    ccCalib2VertexFeatureResult& result,
    ccDiagObject* diagObj=0,
    c_UInt32 diagFlags=0
  );
  ```

  Extracts calibration features from <*srcImage*> based on the specified params.
  ```

Parameters

srcImg	The image containing the calibration plate. The size of checkerboard squares in the image must be at least 15x15 pixels. If Data Matrix codes are used as calibration fiducials, the Data Matrix codes should have a minimum resolution of 4 ppm.
params	The run-time parameters for this function.
result	The result of the extraction.
diagObj	An optional **ccDiagObject**. If you supply a value for diagObj, then the tool will record diagnostic information in the supplied object.
diagFlags	Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

Notes

If **params.labelMode()** is neither *eUseDataMatrix* nor *eUseDataMatrixWithGridPitch*, the vector symbolInfo in <*result*> will be empty.

If **params.labelMode()** is either *eUseDataMatrix* or *eUseDataMatrixWithGridPitch*, the vector symbolInfo in <*result*> will contain an element corresponding to each Data Matrix code that was succesfully found, decoded and parsed by the tool.

The current implementation of the tool stops searching for Data Matrix codes after any single Data Matrix code in the image is succesfully found, decoded and parsed. As a result, the size of this vector will not be greater than one.

If **params.labelMode()** is *eUseDataMatrixWithGridPitch*, then each Data Matrix code on the calibration plate must encode the grid pitch in addition to the grid position. The value of grid pitch encoded in the Data Matrix code is used by the tool, **params.physicalGridPitch()** is ignored.

If **params.labelMode()** is *eUseDataMatrix*, then each Data Matrix code must encode the grid position and may optionally encode the grid pitch. The value of grid pitch encoded in the Data Matrix codes will not be used by the tool, but it will be stored in <*result*>. The tool will use **params.physicalGridPitch()**.

Throws

ccPel::UnboundWindow
: If *srcImg* is not bound to any root image.

ccCalib2VertexFeatureDefs::BadParam
: If client transform of *srcImg* is not linear.

ccCalib2VertexFeatureDefs::BadParam
: If **params.algorithm()** is *eExhaustiveMultiRegion* and **params.labelMode()** is neither *eDataMatrix* nor *eDataMatrixWithGridPitch*

ccCalib2VertexFeatureDefs::FailedCorrespondence
: If consistent labeling of the vertices could not be carried out. This throw may occur due to out-of-focus images or images with severe light gradient or very small checker size in images, and so on.

ccCalib2VertexFeatureDefs::FiducialNotFound
: If running in *ccCalib2VertexFeatureDefs::eUseFiducial* mode and the expected fiducial feature is not found.

ccCalib2VertexFeatureDefs::DataMatrixNotFound
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix could be found in *srcImg*.

ccCalib2VertexFeatureDefs::DataMatrixDecodeError
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix code that was successfully found in *srcImg* could be successfully decoded.

ccCalib2VertexFeatureDefs::DataMatrixParseError
: If running in *ccCalib2VertexFeatureDefs::eUseDataMatrix* mode or *ccCalib2VertexFeatureDefs::eUseDataMatrixWithGridPitch* mode and no Data Matrix code in *srcImg* that was successfully decoded could be successfully parsed.

ccCalib2VertexFeatureDefs::DataMatrixParseError
: If **labelMode()** is *eUseDataMatrixWithGridPitch* and either component of the grid pitch extracted from any Data Matrix code in <*srcImg*> is less than or equal to zero.

cfCalibrateLineScanCamera

```
#include <ch_cvl/ccalibls.h>

cfCalibrateLineScanCamera();
```

Global function to perform calibration of a line scan camera using the supplied image.

cfCalibrateLineScanCamera

```
void cfCalibrateLineScanCamera(
   const cmStd vector<cc2Vect> &imagePoints,
   const cmStd vector<cc2Vect> &physicalPoints,
   const ccCalibrateLineScanCameraParams &params,
   double &rmsResidual, double &maxResidual,
   cc2XformCalib2 &imageFromPhysicalXform);

void cfCalibrateLineScanCamera(
   const ccCrspPairVector &imagePointsPhysicalPoints,
   const ccCalibrateLineScanCameraParams &params,
   double &rmsResidual, double &maxResidual,
   cc2XformCalib2 &imageFromPhysicalXform);
```

- ```
  void cfCalibrateLineScanCamera(
     const cmStd vector<cc2Vect> &imagePoints,
     const cmStd vector<cc2Vect> &physicalPoints,
     const ccCalibrateLineScanCameraParams &params,
     double &rmsResidual, double &maxResidual,
     cc2XformCalib2 &imageFromPhysicalXform);
  ```

  Perform line scan camera calibration using the supplied vectors of corresponded image points and physical points.

  The image from which the image points are extracted must have been acquired from a line scan camera, and the positive y-axis direction must correspond to the direction of apparent camera motion (increasing y-values correspond to successively acquired rows of pixels).

  Parameters

imagePoints	The locations of the calibration plate vertices in image coordinates.
physicalPoints	The physical coordinates of the plate vertices.
params	The calibration parameters.
rmsResidual	The computed RMS residual error across all points.
maxResidual	The maximum residual error for any single point.
  ```

imageFromPhysicalXform
The computed calibration.

Throws

cfCalibrateLineScanCameraDefs::BadParams
The number of physical points differs from the number of image points or *params* specifies that the supplied camera-to-target distance be used, but the supplied distance is either less than or equal to 0 or HUGE_VAL.

ccCalibrateLineScanCameraDefs::InsufficientPoints
Fewer than four points are supplied.

ccMathError::Singular
A transformation cannot be computed. This can occur if all of the supplied points are collinear.

- ```
  void cfCalibrateLineScanCamera(
     const ccCrspPairVector &imagePointsPhysicalPoints,
     const ccCalibrateLineScanCameraParams &params,
     double &rmsResidual, double &maxResidual,
     cc2XformCalib2 &imageFromPhysicalXform);
  ```

Perform line scan camera calibration using the supplied vectors of corresponded image points and physical points.

The image from which the image points are extracted must have been acquired from a line scan camera, and the positive y-axis direction must correspond to the direction of apparent camera motion (increasing y-values correspond to successively acquired rows of pixels).

Parameters

imagePointsPhysicalPoints
A vector of correspondence pairs, where each pair includes an image point and the corresponding physical point. The first point in each correspondence pair must be the image point, the second point the physical point.

params The calibration parameters.

rmsResidual The computed RMS residual error across all points.

maxResidual The maximum residual error for any single point.

imageFromPhysicalXform
The computed calibration.
  ```

Throws

cfCalibrateLineScanCameraDefs::BadParams

The number of physical points differs from the number of image points or *params* specifies that the supplied camera-to-target distance be used, but the supplied distance is either less than or equal to 0 or HUGE_VAL.

ccCalibrateLineScanCameraDefs::InsufficientPoints

Fewer than four points are supplied.

ccMathError::Singular

A transformation cannot be computed. This can occur if all of the supplied points are collinear.

cfCalibrateLineScanCamera

cfCalibrationRun()

```
#include <ch_cvl/calib.h>

cfCalibrationRun();
```

Global function to perform calibration using the supplied image.

cfCalibrationRun

```
void cfCalibrationRun(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccGridCalibParams& params,
  ccGridCalibResults& results);
```

Applies the Calibration tool to the supplied input image using the supplied parameters. The input image must be of a calibration grid that meets the following requirements:

- It must be a grid of circular dots.
- The grid must not be rotated more than +/- 30° with respect to the image coordinate system.
- The grid must contain at least three dots for a linear calibration, ten for a third-order polynomial calibration and 21 for a fifth-order polynomial calibration.
- The dots must be at least 10 pixels in diameter.
- The image contrast between the dots and background must be at least 12.

Parameters

image The input image. This must be an image of a valid calibration grid.

params A **ccGridCalibParams** that specifies the calibration parameters

results A **ccGridCalibResults** into which the calibration results are placed

Throws

ccCalibDefs::GridAngleTooLarge
The grid is rotated excessively with respect to the image coordinate system.

ccCalibDefs::InvalidMap
A valid transformation could not be computed.

ccCalibDefs::TooFewDotsFound
The input image does not contain enough dots.

ccCalibDefs::BadCorrespondence
: The tool could not determine the location and orientation of the grid from the position of the dots.

ccCalibDefs::BadParam
: One or more illegal values were specified in *params*.

Notes

The minimum recommended grid size for the different calibrations are:
eGridLinear: 4x4
ePolynomialOrder3: 8x8
ePolynomialOrder5: 8x8 (12x12 is preferable, if possible)

cfCaliperFindCircle()

```
#include <ch_cvl/clpfind.h>

cfCaliperFindCircle();
```

Global function to find a circle in an image.

cfCaliperFindCircle

```
void cfCaliperFindCircle (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccCaliperFinderBaseRunParams &runParams,
  ccCaliperCircleFinderResult& result);
```

Parameters

image An image containing a circle you wish to find.

runParams A run-time parameters object. Default constructed *runParams* are not valid. You must set parameters such as *numCalipers*, *caliperSize*, and *caliperSampling* to valid values.

result Where to put the results.

Throws

ccCaliperFinderBaseDefs::UnboundImage
If the image is unbound.

ccCaliperFinderBaseDefs::BadParams
If the *runParams* and *result* are not for the same shape,
or if *runParams* is invalid,
or if the *runParams* sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

Notes

Any failure as the caliper runs will mark the edge as not found. For example, if the *caliperRunParams* is set to **autoClip(false)** and the affine rectangle extends past the image, the edge will be marked as not found.

The regions (affine rectangles) where we apply the calipers depend upon the start pose. Each caliper has a nominal region. At run time, that nominal region is mapped by the start pose to determine the actual region where the caliper is applied.

If your application processes many images in a loop, the finder tool will run faster if you reuse the same caliper result set object because some information is cached in this object.

Only the highest scoring edge of each caliper is passed to the fitter tool. Therefore, be certain that the applied calipers will only find edges corresponding to the desired feature. If the caliper finds an unexpected edge, the results may be inaccurate.

cfCaliperFindEllipse()

```
#include <ch_cvl/clpfind.h>

cfCaliperFindEllipse();
```

Global function to find a ellipse in an image.

cfCaliperFindEllipse

```
void cfCaliperFindEllipse (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccCaliperFinderBaseRunParams &runParams,
  ccCaliperEllipseFinderResult& result);
```

Parameters

image	An image containing a ellipse you wish to find.
runParams	A run-time parameters object. Default constructed *runParams* are not valid. You must set parameters such as *numCalipers*, *caliperSize*, and *caliperSampling* to valid values.
result	Where to put the results.

Throws

ccCaliperFinderBaseDefs::UnboundImage
If the image is unbound.

ccCaliperFinderBaseDefs::BadParams
If the *runParams* and *result* are not for the same shape,
or if *runParams* is invalid,
or if the *runParams* sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

Any failure as the caliper runs will mark the edge as not found. For example, if the *caliperRunParams* is set to **autoClip(false)** and the affine rectangle extends past the image, the edge will be marked as not found.

The regions (affine rectangles) where we apply the calipers depend upon the start pose. Each caliper has a nominal region. At run time, that nominal region is mapped by the start pose to determine the actual region where the caliper is applied.

If your application processes many images in a loop, the finder tool will run faster if you reuse the same caliper result set object because some information is cached in this object.

Only the highest scoring edge of each caliper is passed to the fitter tool. Therefore, be certain that the applied calipers will only find edges corresponding to the desired feature. If the caliper finds an unexpected edge, the results may be inaccurate.

cfCaliperFindLine()

```
#include <ch_cvl/clpfind.h>

cfCaliperFindLine();
```

Global function to find a line in an image.

cfCaliperFindLine

```
void cfCaliperFindLine (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccCaliperFinderBaseRunParams &runParams,
  ccCaliperLineFinderResult& result);
```

Parameters

image	An image containing a line you wish to find.
runParams	A run-time parameters object. Default constructed *runParams* are not valid. You must set parameters such as *numCalipers*, *caliperSize*, and *caliperSampling* to valid values.
result	Where to put the results.

Throws

ccCaliperFinderBaseDefs::UnboundImage
If the image is unbound.

ccCaliperFinderBaseDefs::BadParams
If the *runParams* and *result* are not for the same shape,
or if *runParams* is invalid,
or if the *runParams* sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

Any failure as the caliper runs will mark the edge as not found. For example, if the *caliperRunParams* is set to **autoClip(false)** and the affine rectangle extends past the image, the edge will be marked as not found.

The regions (affine rectangles) where we apply the calipers depend upon the start pose. Each caliper has a nominal region. At run time, that nominal region is mapped by the start pose to determine the actual region where the caliper is applied.

If your application processes many images in a loop, the finder tool will run faster if you reuse the same caliper result set object because some information is cached in this object.

Only the highest scoring edge of each caliper is passed to the fitter tool. Therefore, be certain that the applied calipers will only find edges corresponding to the desired feature. If the caliper finds an unexpected edge, the results may be inaccurate.

cfCaliperFindShape()

```
#include <ch_cvl/clpfind.h>

cfCaliperFindShape();
```

Global function to find a shape in an image.

You can use this function as an alternative to **cfCaliperFindLine()**, **cfCaliperFindCircle()**, or **cfCaliperFindEllipse()**. For example, if you call **cfCaliperFindShape()** with a circle run-time parameters object and a circle results object, the function will attempt to find a circle in the image.

cfCaliperFindShape

```
void cfCaliperFindShape (
  const ccPelBuffer_const<c_UInt8>& image,
  const ccCaliperFinderBaseRunParams &runParams,
  ccCaliperShapeFinderResult& result);
```

Parameters

image	An image containing a shape you wish to find.
runParams	A run-time parameters object. Default constructed *runParams* are not valid. You must set parameters such as *numCalipers*, *caliperSize*, and *caliperSampling* to valid values.
result	Where to put the results.

Throws

ccCaliperFinderBaseDefs::UnboundImage
If the image is unbound.

ccCaliperFinderBaseDefs::BadParams
If the *runParams* and *result* are not for the same shape,
or if *runParams* is invalid,
or if the *runParams* sampling rate causes an overflow,
or if the resulting number of samples is less than (1, 1).

Any failure as the caliper runs will mark the edge as not found. For example, if the *caliperRunParams* is set to **autoClip(false)** and the affine rectangle extends past the image, the edge will be marked as not found.

The regions (affine rectangles) where we apply the calipers depend upon the start pose. Each caliper has a nominal region. At run time, that nominal region is mapped by the start pose to determine the actual region where the caliper is applied.

If your application processes many images in a loop, the finder tool will run faster if you reuse the same caliper result set object because some information is cached in this object.

Only the highest scoring edge of each caliper is passed to the fitter tool. Therefore, be certain that the applied calipers will only find edges corresponding to the desired feature. If the caliper finds an unexpected edge, the results may be inaccurate.

cfCaliperRun()

```
#include <ch_cvl/caliper.h>

cfCaliperRun();
```

Global function to apply the Caliper tool to an input image.

cfCaliperRun

```
void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
  const ccAffineSamplingParams& affSamplParams,
  const ccCaliperRunParams& runParams,
  ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0);

void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
  const ccCaliperProjectionParams& projParams,
  const ccCaliperRunParams& runParams,
  ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0);

void cfCaliperRun(
  const ccPelBuffer_const<c_UInt32>& projImage,
  const c_UInt32 nPelsPerBin,
  const cc2Vect& clientOrigin,
  const ccCaliperRunParams& runParams,
  ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0);

void cfCaliperRun(
  const ccPelBuffer_const<c_UInt32>& projImage,
  const ccPelBuffer_const<c_UInt32>& weights,
  const cc2Vect& clientOrigin,
  const ccCaliperRunParams& runParams,
  ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
  c_UInt32 dFlags=0);

void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
  const cmStd vector<ccAffineSamplingParams>&
  affSamplParams,
```

```
  const cmStd vector<ccCaliperRunParams>& runParams,
  cmStd vector<ccCaliperResultSet>& resultSet,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);

void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
  const ccAffineSamplingParams& affSamplParams,
  const ccCaliperCorrelationRunParams& runParams,
  ccCaliperCorrelationResultSet& resultSet,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);

void cfCaliperRun(
  const ccPelBuffer_const<c_UInt32>& projImage,
  const c_UInt32 nPelsPerBin, const cc2Vect& clientOrigin,
  const ccCaliperCorrelationRunParams& runParams,
  ccCaliperCorrelationResultSet& resultSet,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);

void cfCaliperRun(
  const ccPelBuffer_const<c_UInt32>& projImage,
  const ccPelBuffer_const<c_UInt32>& weights,
  const cc2Vect& clientOrigin,
  const ccCaliperCorrelationRunParams& runParams,
  ccCaliperCorrelationResultSet& resultSet,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);

void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
  const cmStd vector<ccAffineSamplingParams>&
  affSamplParams,
  const cmStd vector<ccCaliperCorrelationRunParams>&
  runParams,
  cmStd vector<ccCaliperCorrelationResultSet>& resultSet,
  ccDiagObject* dObj=0, c_UInt32 dFlags=0);
```

- ```
  void cfCaliperRun (
    const ccPelBuffer_const<c_UInt8>& image,
    const ccAffineSamplingParams& affSamplParams,
    const ccCaliperRunParams& runParams,
    ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied input image using the supplied parameters.

  Parameters

image	The input image
affSamplParams	The projection region within *image*
runParams	A **ccCaliperRunParams**
  ```

resultSet A reference to a **ccCaliperResultSet** into which the results are placed

dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void cfCaliperRun (
    const ccPelBuffer_const<c_UInt8>& image,
    const ccCaliperProjectionParams& projParams,
    const ccCaliperRunParams& runParams,
    ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied input image using the supplied parameters.

  Parameters

  image The input image

  projParams The projection region within *image*

  runParams A **ccCaliperRunParams**

  resultSet A reference to a **ccCaliperResultSet** into which the results are placed

  dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

  dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

  ccDiagDefs::eInputs
  ccDiagDefs::eIntermediate
  ccDiagDefs::eResults

  with one of the following values:
  ```

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Notes

You should not use this overload. Instead, specify the projection region using a **ccAffineSamplingParams** by calling the next overload.

- ```
  void cfCaliperRun (
    const ccPelBuffer_const<c_UInt32>& projImage,
    const c_UInt32& nPelsPerBin, const cc2Vect& clientOrigin,
    const ccCaliperRunParams& runParams,
    ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied projection image using the supplied parameters. Use this overload if you have already created a one-dimensional projection image in which you want to locate and score edges.

  Parameters

projImage	The projection image
nPelsPerBin	The number of pixels which were summed to produce the values in each pixel of *projImage*
clientOrigin	The location with respect to which results are to be returned
runParams	A **ccCaliperRunParams**
resultSet	A reference to a **ccCaliperResultSet** into which the results are placed
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*
  ```

- ```
  void cfCaliperRun(
    const ccPelBuffer_const<c_UInt32>& projImage,
    const ccPelBuffer_const<c_UInt32>& weights,
    const cc2Vect& clientOrigin,
    const ccCaliperRunParams& runParams,
    ccCaliperResultSet& resultSet, ccDiagObject* dObj=0,
    c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied projection image using the supplied parameters. Use this overload if not all pixels in the projection image were summed using the same number of pixels; you supply a weights image that contains the number of pixels that were summed to produce the corresponding pixel in the projection image.

  Parameters

projImage	The projection image
weights	The weights image
clientOrigin	The location with respect to which results are to be returned
runParams	A **ccCaliperRunParams**
resultSet	A reference to a **ccCaliperResultSet** into which the results are placed
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

- ```
  void cfCaliperRun (
    const ccPelBuffer_const<c_UInt8>& image,
    const vector<ccAffineSamplingParams>& affSamplParams,
  ```

```
const vector<ccCaliperRunParams>& runParams,
vector<ccCaliperResultSet>& resultSet,
ccDiagObject* dObj=0, c_UInt32 dFlags=0);
```

Applies the Caliper tool to the supplied input image multiple times, once for each element of the supplied vector of **ccAffineSamplingParams**. The number of elements in the supplied vector of **ccAffineSamplingParams** *must* match the number of elements in the supplied vector of **ccCaliperResultSet**.

If you supply a vector of **ccCaliperRunParams** with a single element, then the same **ccCaliperRunParams** will be used for each application of the Caliper tool. If the supplied vector of **ccCaliperRunParams** contains more than 1 element, it must contain the same number of elements as the supplied vectors of **ccAffineSamplingParams** and **ccCaliperResultSet**.

Parameters

image The input image

affSamplParams A vector of **ccAffineSamplingParams** that specify the projection regions within *image*

runParams A vector of **ccCaliperRunParams**

resultSet A vector to a **ccCaliperResultSet** into which the results are placed

dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccCaliperDefs::BadParams

If the number of elements in *runParams* is 1, then the number of elements in *affSamplParams* must match the number of elements in *resultSet*. If the number of elements in *runParams* is greater than 1, then the number of elements in *runParams* must match the number of elements in *affSamplParams* and *resultSet*.

- ```
  void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
    const ccAffineSamplingParams& affSamplParams,
    const ccCaliperCorrelationRunParams& runParams,
    ccCaliperCorrelationResultSet& resultSet,
    ccDiagObject* dObj=0, c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied input image using the supplied correlation-mode parameters.

  Parameters

image	The input image
affSamplParams	The projection region within *image*
runParams	A **ccCaliperCorrelationRunParams**
resultSet	A reference to a **ccCaliperCorrelationResultSet** into which the results are placed
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

- ```
  void cfCaliperRun(
    const ccPelBuffer_const<c_UInt32>& projImage,
    const c_UInt32 nPelsPerBin, const cc2Vect& clientOrigin,
    const ccCaliperCorrelationRunParams& runParams,
    ccCaliperCorrelationResultSet& resultSet,
    ccDiagObject* dObj=0, c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied projection image using the supplied correlation-mode parameters. Use this overload if you have already created a one-dimensional projection image in which you want to locate a pattern.

Parameters

projImage	The projection image
nPelsPerBin	The number of pixels which were summed to produce the values in each pixel of *projImage*
clientOrigin	The location with respect to which results are to be returned
runParams	A **ccCaliperCorrelationRunParams**
resultSet	A reference to a **ccCaliperCorrelationResultSet** into which the results are placed
dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

- ```
  void cfCaliperRun(
    const ccPelBuffer_const<c_UInt32>& projImage,
    const ccPelBuffer_const<c_UInt32>& weights,
    const cc2Vect& clientOrigin,
    const ccCaliperCorrelationRunParams& runParams,
    ccCaliperCorrelationResultSet& resultSet,
    ccDiagObject* dObj=0, c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied projection image using the supplied correlation-mode parameters. Use this overload if not all pixels in the projection image were summed using the same number of pixels; you supply a weights image that contains the number of pixels that were summed to produce the corresponding pixel in the projection image.

  Parameters

projImage	The projection image
weights	The weights image
clientOrigin	The location with respect to which results are to be returned
  ```

runParams A **ccCaliperCorrelationRunParams**

resultSet A reference to a **ccCaliperCorrelationResultSet** into which the results are placed

dObj An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.

dFlags The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

- ```
  void cfCaliperRun(const ccPelBuffer_const<c_UInt8>& image,
    const cmStd vector<ccAffineSamplingParams>&
    affSamplParams,
    const cmStd vector<ccCaliperCorrelationRunParams>&
    runParams,
    cmStd vector<ccCaliperCorrelationResultSet>& resultSet,
    ccDiagObject* dObj=0, c_UInt32 dFlags=0);
  ```

  Applies the Caliper tool to the supplied input image multiple times, once for each element of the supplied vector of **ccAffineSamplingParams**. The number of elements in the supplied vector of **ccAffineSamplingParams** *must* match the number of elements in the supplied vector of **ccCaliperCorrelationResultSet**.

  If you supply a vector of **ccCaliperCorrelationRunParams** with a single element, then the same **ccCaliperCorrelationRunParams** will be used for each application of the Caliper tool. If the supplied vector of **ccCaliperCorrelationRunParams** contains more than 1 element, it must contain the same number of elements as the supplied vectors of **ccAffineSamplingParams** and **ccCaliperCorrelationResultSet**.

  Parameters

  image The input image

  affSamplParams A vector of **ccAffineSamplingParams** that specify the projection regions within *image*

  runParams A vector of **ccCaliperCorrelationRunParams**

  resultSet A vector to a **ccCaliperCorrelationResultSet** into which the results are placed
  ```

cfCaliperRun()

dObj	An optional **ccDiagObject**. If you supply a value for *dObj*, then the tool will record diagnostic information in the supplied object.
dFlags	The optional diagnostic flags. *dFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

cfCircleFit()

```
#include <ch_cvl/fit.h>

cfCircleFit();
```

Global function to fit a circle to a set of points.

cfCircleFit

```
void cfCircleFit(
  const ccCircleFitParams &params,
  cmStd vector<cc2Vect> &pts,
  ccCircleFitResults &results);
```

A global function that fits a circle to the supplied set of points using the supplied **ccCircleFitParams**.

Parameters

params — A **ccCircleFitParams** describing the fitting method, number of outliers to ignore, and maximum RMS error threshold.

pts — A vector of points specified in a single coordinate system.

results — A reference to a **ccCircleFitResults** into which the results will be placed. Any existing contents of *results* are cleared.

Throws

ccMathError::Singular
The number of points in *pts* is less than 3, after subtracting the number of points to ignore specified in params,
or *pts* does not include three points which are not colinear.

cfCircleFit()

cfColorMatch()

```
#include <ch_cvl/colmatch.h>

class ccColorMatchResult;
```

Calculates the results of a color match between an image (or single color) and a set of reference colors.

cfColorMatch

```
void cfColorMatch(const ccColorValue& color,
       const cmStd vector<ccColorValue>& referenceColors,
       const ccColorMatchRunParams &runParams,
       ccColorMatchResult &result);

void cfColorMatch(const cc3PlanePelBuffer& image,
       const cmStd vector<ccColorValue>& referenceColors,
       const ccColorMatchRunParams &runParams,
       ccColorMatchResult &result);

void cfColorMatch(const cc3PlanePelBuffer& image,
       const ccShapePtrh& region,
       const cmStd vector<ccColorValue>& referenceColors,
       const ccColorMatchRunParams &runParams,
       ccColorMatchResult &result);

void cfColorMatch(const cc3PlanePelBuffer& image,
       const ccPelBuffer_const<c_UInt8>& mask,
       const cmStd vector<ccColorValue>& referenceColors,
       const ccColorMatchRunParams &runParams,
       ccColorMatchResult &result);
```

- ```
  void cfColorMatch(const ccColorValue& color,
         const cmStd vector<ccColorValue>& referenceColors,
         const ccColorMatchRunParams &runParams,
         ccColorMatchResult &result);
  ```

  Performs a color match between *color* and *referenceColors*.

  If necessary the color spaces of *color* and *referenceColors* are converted to the same color space as *runParams*.

  Parameters

color	The color.
referenceColors	The reference colors.
runParams	A **ccColorMatchRunParams** object that specifies the color space, distance metric, and weights to perform the match.
  ```

result A **ccColorMatchResult** object that contains the results of performing the match.

Throws

ccColorMatchDefs::BadParams
referenceColors is empty

ccColorMatchDefs::BadColorSpace
either the color space of *color* or *referenceColors* could not be converted to the same color space as *runParams*.

ccColorMatchDefs::BadColorSpace
referenceColors has the color space **ccColorSpaceDefs::eUnknown**.

- ```
  void cfColorMatch(const cc3PlanePelBuffer& image,
         const cmStd vector<ccColorValue>& referenceColors,
         const ccColorMatchRunParams &runParams,
         ccColorMatchResult &result);
  ```

Uses **cfGetSimpleColorFromImage()** to compute the mean color from *image* and matches it with the collection of colors in *referenceColors*.

If necessary the color spaces of *color* and *referenceColors* are converted to the same color space as *runParams*.

Parameters

image The image.

referenceColors The reference colors.

runParams A **ccColorMatchRunParams** object that specifies the color space, distance metric, and weights to perform the match.

result A **ccColorMatchResult** object that contains the results of performing the match. If the size of *referenceColors* is 1, the **ccColorMatchResult**::**confidenceScore()** is 1.0 and **ccColorMatchResult**::**label()** is 0.

Throws

ccColorMatchDefs::BadParams
referenceColors is empty

ccColorMatchDefs::BadColorSpace
either the color space of *color* or *referenceColors* could not be converted to the same color as *runParams*.
  ```

ccColorMatchDefs::BadColorSpace
referenceColors has the color space **ccColorSpaceDefs::eUnknown**.

ccColorMatchDefs::NotEnoughSamples
Not enough pels (< 1) in the image satisfy the constraints of *runParams*.

- ```
  void cfColorMatch(const cc3PlanePelBuffer& image,
        const ccShapePtrh& region,
        const cmStd vector<ccColorValue>& referenceColors,
        const ccColorMatchRunParams &runParams,
        ccColorMatchResult &result);
  ```

Uses **cfGetSimpleColorFromImageRegion()** to compute the mean color from the region of *image* described by *region* and matches it with the collection of colors in *referenceColors*.

Region is defined in image's client coordinate system and can partially overlap *image*. In that case the intersection is used. If more than 50% of a pel falls inside the region, it is considered as being completely inside the region

If necessary the color spaces of *color* and *referenceColors* are converted to the same color space as *runParams*.

Parameters

image The image.

region A shape that defines a region. If *region* is NULL, the whole image is used.

referenceColors The reference colors.

runParams A **ccColorMatchRunParams** object that specifies the color space, distance metric, and weights to perform the match.

result A **ccColorMatchResult** object that contains the results of performing the match. If the size of *referenceColors* is 1, the **ccColorMatchResult**::**confidenceScore()** is 1.0 and **ccColorMatchResult**::**label()** is 0.

Throws

ccColorMatchDefs::BadParams
referenceColors is empty

ccColorMatchDefs::BadColorSpace
either the color space of *color* or *referenceColors* could not be converted to the same color space as *runParams*.
  ```

ccColorMatchDefs::BadColorSpace
referenceColors has the color space **ccColorSpaceDefs::eUnknown**.

ccColorMatchDefs::NotEnoughSamples
region and *image* do not intersect

Not enough pels (< 1) in the image satisfy the constraints of *runParams* or *region*.

ccColorMatchDefs::ShapeIsNotRegion
region is not a region shape.

ccColorMatchDefs::NonLinearXform
the image from client transform is nonlinear

- ```
  void cfColorMatch(const cc3PlanePelBuffer& image,
         const ccPelBuffer_const<c_UInt8>& mask,
         const cmStd vector<ccColorValue>& referenceColors,
         const ccColorMatchRunParams &runParams,
         ccColorMatchResult &result);
  ```

Uses **cfGetSimpleColorFromImage()** to compute the mean color from *image* (using hte supplied *mask*) and matches it with the collection of colors in *referenceColors*.

If necessary the color spaces of *color* and *referenceColors* are converted to the same color space as *runParams*.

Parameters

image The image.

mask An image mask. Pels whose value is 0 are don't care pixels; pels whose values are 255 are care pixels. All other values throw an error.

referenceColors The reference colors.

runParams A **ccColorMatchRunParams** object that specifies the color space, distance metric, and weights to perform the match.

result A **ccColorMatchResult** object that contains the results of performing the match. If the size of *referenceColors* is 1, the **ccColorMatchResult**::**confidenceScore()** is 1.0 and **ccColorMatchResult**::**label()** is 0.

Throws

ccColorMatchDefs::BadParams
referenceColors is empty
  ```

ccColorMatchDefs::BadColorSpace
either the color space of *color* or *referenceColors* could not be converted to the same color space as *runParams*.

ccColorMatchDefs::BadColorSpace
referenceColors has the color space **ccColorSpaceDefs::eUnknown**.

ccColorMatchDefs::NotEnoughSamples
Not enough pels (< 1) in the image satisfy the constraints of *mask* or *runParams*.

cfColorMatch()

cfConvertCDBtoVDB()

```
#include <ch_cvl/cdb.h>

cfConvertCDBtoVDB();
```

Global function that converts a CDB file to a VDB file.

cfConvertCDBtoVDB

```
cfConvertCDBtoVDB(const ccCvlString &cdbFile,
  const ccCvlString &vdbFile,
  ccCvlOStream *out = 0);
```

Converts CDB file named *cdbFile* to the VDB file named *vdbFile*.

Parameters

cdbFile	The name of the CDB file to convert.
vdbFile	The name of the VDB file to write.
out	An optional output stream to report errors and diagnostic messages.

Notes

Only records of *IMAGE_TYPE* and *ACQUIRE_TYPE* are written into destination file.

cfConvertCDBtoVDB()

cfConvertDisplayFormat2ImageFormat()

```
#include <ch_cvl/display.h>

cfConvertDisplayFormat2ImageFormat();
```

Global function to convert **DisplayFormat** values to their corresponding **ceImageFormat** values.

cfConvertDisplayFormat2ImageFormat

```
cfConvertDisplayFormat2ImageFormat(ccDisplay::DisplayFormat);
```

Converts the specified **DisplayFormat** value to its corresponding **ceImageFormat** value.

Notes

This function is intended to be used when constructing **ccAcqFifo**s that will be used exclusively with **startLiveDisplay()**.

Example

The following code fragment illustrates the use of this function.

```
// Determine the acquisition image format that matches the
// current display settings.
ceImageFormat ideal =
  cfConvertDisplayFormat2ImageFormat(myDisplay.displayFormat());

// Create a FIFO to acquire in the ideal format, if supported
ccAcqFifoPtrh fifo;
if (fg.isSupportedEx(vidfmt, ideal))
    fifo = vidfmt->newAcqFifoEx(fg, ideal);

// Otherwise, acquire in the format specified in the video format
else
  fifo = vidfmt->newAcqFifoEx(fg);

// Display a live image using the newly created fifo.
myDisplay.startLiveDisplay(fifo);
```

cfConvertDisplayFormat2ImageFormat()

cfConvertPel()

```
#include <ch_cvl/colorpel.h>

cfConvertPel();
```

Global function to convert pixels of one type to another.

cfConvertPel

```
void cfConvertPel (ccPackedRGB15Pel& t,
 const c_UInt8& p)

void cfConvertPel (ccPackedRGB15Pel& t,
 const ccPackedRGB15Pel& p)

void cfConvertPel (ccPackedRGB15Pel& t,
 const ccPackedRGB16Pel& p)

void cfConvertPel (ccPackedRGB15Pel& t,
 const ccPackedRGB24Pel& p)

void cfConvertPel (ccPackedRGB15Pel& t,
 const ccPackedRGB32Pel& p)

void cfConvertPel (ccPackedRGB16Pel& t, const c_UInt8& p)

void cfConvertPel (ccPackedRGB16Pel& t,
 const ccPackedRGB15Pel& p)

void cfConvertPel (ccPackedRGB16Pel& t,
 const ccPackedRGB16Pel& p)

void cfConvertPel (ccPackedRGB16Pel& t,
 const ccPackedRGB24Pel& p)

void cfConvertPel (ccPackedRGB16Pel& t,
 const ccPackedRGB32Pel& p)

void cfConvertPel (ccPackedRGB24Pel& t, const c_UInt8& f)

void cfConvertPel (ccPackedRGB24Pel& t,
 const ccPackedRGB15Pel& p)

void cfConvertPel (ccPackedRGB24Pel& t,
 const ccPackedRGB16Pel& p)

void cfConvertPel (ccPackedRGB24Pel& t,
 const ccPackedRGB24Pel& p)

void cfConvertPel (ccPackedRGB24Pel& t,
 const ccPackedRGB32Pel& p)
```

```
void cfConvertPel (ccPackedRGB32Pel& t, const c_UInt8& p)

void cfConvertPel (ccPackedRGB32Pel& t,
 const ccPackedRGB15Pel& p)

void cfConvertPel (ccPackedRGB32Pel& t,
 const ccPackedRGB16Pel& p)

void cfConvertPel (ccPackedRGB32Pel& t,
 const ccPackedRGB24Pel& p)

void cfConvertPel (ccPackedRGB32Pel& t,
 const ccPackedRGB32Pel& p)

void cfConvertPel (c_UInt8& t, const c_UInt8& f)

void cfConvertPel (c_UInt8& t, const ccPackedRGB15Pel& f)

void cfConvertPel (c_UInt8& t, const ccPackedRGB16Pel& f)

void cfConvertPel (c_UInt8& t, const ccPackedRGB24Pel& p)

void cfConvertPel (c_UInt8& t, const ccPackedRGB32Pel& f)
```

- `void cfConvertPel (ccPackedRGB15Pel& t, const c_UInt8& p)`

  Converts the 8-bit pel *p* to a **ccPackedRGB15Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- `void cfConvertPel (ccPackedRGB15Pel& t, const ccPackedRGB15Pel& p)`

  Converts the **ccPackedRGB15Pel** pel *p* to a **ccPackedRGB15Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- `void cfConvertPel (ccPackedRGB15Pel& t, const ccPackedRGB16Pel& p)`

  Converts the **ccPackedRGB16Pel** pel *p* to a **ccPackedRGB15Pel** and copies it to *t*.

Parameters

to The converted pixel.

from The pixel to conver.

- `void cfConvertPel (ccPackedRGB15Pel& t, const ccPackedRGB24Pel& p)`

  Converts the **ccPackedRGB24Pel** pel *p* to a **ccPackedRGB15Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB15Pel& t, const ccPackedRGB32Pel& p)`

  Converts the **ccPackedRGB32Pel** pel *p* to a **ccPackedRGB15Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB16Pel& t, const c_UInt8& p)`

  Converts the 8-bit pel *p* to a **ccPackedRGB16Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB16Pel& t, const ccPackedRGB15Pel& p)`

  Converts the **ccPackedRGB15Pel** pel *p* to a **ccPackedRGB16Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB16Pel& t, const ccPackedRGB16Pel& p)`

  Converts the **ccPackedRGB16Pel** pel *p* to a **ccPackedRGB16Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

from The pixel to conver.

- `void cfConvertPel (ccPackedRGB16Pel& t, const ccPackedRGB24Pel& p)`

  Converts the **ccPackedRGB24Pel** pel *p* to a **ccPackedRGB16Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB16Pel& t, const ccPackedRGB32Pel& p)`

  Converts the **ccPackedRGB32Pel** pel *p* to a **ccPackedRGB16Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB24Pel& t, const c_UInt8& f)`

  Converts the 8-bit pel *p* to a **ccPackedRGB24Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB24Pel& t, const ccPackedRGB15Pel& p)`

  Converts the **ccPackedRGB15Pel** pel *p* to a **ccPackedRGB24Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- `void cfConvertPel (ccPackedRGB24Pel& t, const ccPackedRGB16Pel& p)`

  Converts the **ccPackedRGB16Pel** pel *p* to a **ccPackedRGB24Pel** and copies it to *t*.

  Parameters

  to The converted pixel.

  from The pixel to conver.

- ```
  void cfConvertPel (ccPackedRGB24Pel& t,
    const ccPackedRGB24Pel& p)
  ```

  Converts the **ccPackedRGB24Pel** pel *p* to a **ccPackedRGB24Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (ccPackedRGB24Pel& t,
    const ccPackedRGB32Pel& p)
  ```

  Converts the **ccPackedRGB32Pel** pel *p* to a **ccPackedRGB24Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (ccPackedRGB32Pel& t, const c_UInt8& p)
  ```

  Converts the 8-bit pel *p* to a **ccPackedRGB32Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (ccPackedRGB32Pel& t,
    const ccPackedRGB15Pel& p)
  ```

  Converts the **ccPackedRGB15Pel** pel *p* to a **ccPackedRGB32Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (ccPackedRGB32Pel& t,
    const ccPackedRGB16Pel& p)
  ```

  Converts the **ccPackedRGB16Pel** pel *p* to a **ccPackedRGB32Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.
  ```

- ```
  void cfConvertPel (ccPackedRGB32Pel& t,
    const ccPackedRGB24Pel& p)
  ```

  Converts the **ccPackedRGB24Pel** pel *p* to a **ccPackedRGB32Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (ccPackedRGB32Pel& t,
    const ccPackedRGB32Pel& p)
  ```

  Converts the **ccPackedRGB32Pel** pel *p* to a **ccPackedRGB32Pel** and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (c_UInt8& t, const c_UInt8& f)
  ```

  Converts the 8-bit pel *p* to an 8-bit pel and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (c_UInt8& t, const ccPackedRGB15Pel& f)
  ```

  Converts the **ccPackedRGB15Pel** pel *p* to an 8-bit pel and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (c_UInt8& t, const ccPackedRGB16Pel& f)
  ```

  Converts the **ccPackedRGB16Pel** pel *p* to an 8-bit pel and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

- ```
  void cfConvertPel (c_UInt8& t, const ccPackedRGB24Pel& p)
  ```

  Converts the **ccPackedRGB24Pel** pel *p* to an 8-bit pel and copies it to *t*.

Parameters

to	The converted pixel.
from	The pixel to conver.

- `void cfConvertPel (c_UInt8& t, const ccPackedRGB32Pel& f)`

  Converts the **ccPackedRGB32Pel** pel p to an 8-bit pel and copies it to *t*.

  Parameters

to	The converted pixel.
from	The pixel to conver.

cfConvertPel()

cfConvertString()

```
#include <ch_cvl/unicode.h>

cfConvertString();
```

Global function to convert string between ANSI and Unicode formats. All of these functions throw *ccBadUnicodeChar* if a Unicode character cannot be converted to ANSI.

cfConvertString

```
void cfConvertString (cmStd string& to,
  const cmStd wstring& from);

void cfConvertString (cmStd wstring& to,
  const cmStd string& from);

void cfConvertString (cmStd wstring& to,
  int from);

static inline void cfConvertString (cmStd wstring& to,
  const cmStd wstring& from);

void cfConvertString (cmStd string& to, int from);

static inline void cfConvertString (cmStd string& to,
  const cmStd string& from);

void cfConvertString (char* to, const wchar_t* from);

void cfConvertString (wchar_t* to, const char* from);

static inline void cfConvertString (char* to,
  const char* from);

static inline void cfConvertString (wchar_t* to,
  const wchar_t* from);

void cfConvertString (wchar_t* to, const char* from,
  int len);

static inline void cfConvertString (char* to,
  const char* from, int len);

static inline void cfConvertString (wchar_t* to,
  const wchar_t* from, int len)
```

- ```
  void cfConvertString (cmStd string& to,
    const cmStd wstring& from);
  ```

  Converts and copies the contents of the Unicode string *from* to the ANSI string *to*.
  ```

Parameters

to	The ANSI string to receive the converted string.
from	The Unicode string to convert to ANSI.

- ```
  void cfConvertString (cmStd wstring& to,
    const cmStd string& from);
  ```

  Converts and copies the contents of the ANSI string *from* to the Unicode string *to*.

  Parameters

to	The Unicode string to receive the converted string.
from	The ANSI string to convert to Unicode.

- ```
  void cfConvertString (cmStd wstring& to, int from);
  ```

  Converts integer *from* into characters and places the result in the Unicode string *to*.

  Parameters

to	The Unicode string to receive the converted integer.
from	The integer string to convert to Unicode characters.

- ```
  static inline void cfConvertString (cmStd wstring& to,
    const cmStd wstring& from);
  ```

  Copies the Unicode string *from* to the Unicode string *to*.

  Parameters

to	The Unicode string to receive the copy.
from	The Unicode string to be copied.

- ```
  void cfConvertString (cmStd string& to, int from);
  ```

  Converts integer *from* into characters and places the result in the ANSI string *to*.

  Parameters

to	The ANSI string to receive the converted integer.
from	The integer string to convert to ANSI characters.

- `static inline void cfConvertString (cmStd string& to, const cmStd string& from);`

  Copies the ANSI string *from* to the ANSI string *to*.

  Parameters

to	The ANSI string to receive the copy.
from	The ANSI string to be copied.

- `void cfConvertString (char* to, const wchar_t* from);`

  Converts and copies the null-terminated Unicode character vector *from* to the null-terminated ANSI character vector *to*.

  Parameters

to	The ANSI character vector to receive the copy. The vector must be large enough to hold all the characters in *from*.
from	The Unicode character vector to be copied.

- `void cfConvertString (wchar_t* to, const char* from);`

  Converts and copies the null-terminated ANSI character vector *from* to the null-terminated Unicode character vector *to*.

  Parameters

to	The Unicode character vector to receive the copy. The vector must be large enough to hold all the characters in *from*.
from	The ANSI character vector to be copied.

- `static inline void cfConvertString (char* to, const char* from);`

  Copies the null-terminated ANSI character vector *from* to the null-terminated ANSI character vector *to*.

  Parameters

to	The ANSI character vector to receive the copy. The vector must be large enough to hold all the characters in *from*.
from	The ANSI character vector to be copied.

cfConvertString()

- ```
  static inline void cfConvertString (wchar_t* to,
    const wchar_t* from);
  ```

  Copies the null-terminated Unicode character vector *from* to the null-terminated Unicode character vector *to*.

  Parameters

to	The Unicode character vector to receive the copy. The vector must be large enough to hold all the characters in *from*.
from	The Unicode character vector to be copied.

- ```
  void cfConvertString (wchar_t* to, const char* from,
    int len);
  ```

  Converts and copies up to *len* characters of the null-terminated ANSI character vector *from* to the null-terminated Unicode character vector *to*.

  Parameters

to	The Unicode character vector to receive the copy. The vector must be large enough to hold *len* characters.
from	The ANSI character vector to be copied.
len	The maximum number of characters to copy.

- ```
  static inline void cfConvertString (char* to,
    const char* from, int len);
  ```

  Converts and copies up to *len* characters of the null-terminated ANSI character vector *from* to the null-terminated ANSI character vector *to*.

  Parameters

to	The ANSI character vector to receive the copy. The vector must be large enough to hold *len* characters.
from	The ANSI character vector to be copied.
len	The maximum number of characters to copy.

- ```
  static inline void cfConvertString (wchar_t* to,
    const wchar_t* from, int len);
  ```

  Copies up to *len* characters of the null-terminated Unicode character vector *from* to the null-terminated Unicode character vector *to*.

Parameters

to	The Unicode character vector to receive the copy. The vector must be large enough to hold *len* characters.
from	The Unicode character vector to be copied.
len	The maximum number of characters to copy.

cfConvertString()

cfConvertToFeaturelet()

```
#include <ch_cvl/edge.h>

cfConvertToFeaturelet();
```

Global function that converts an edgelet to a featurelet. Edgelets are described in the **ccEdgelet** reference page and featurelets are described in the **ccFeaturelet** reference page.

cfConvertToFeaturelet

```
static inline ccFeaturelet cfConvertToFeaturelet(
  const ccEdgelet &edgelet);
```

Returns a featurelet corresponding to the given edgelet.

The featurelet will have the edgelet's position, angle, and magnitude. The featurelet's *weight* is set to 1 and *isMod180* is set to false.

Parameters

edgelet	The edgelet to convert.

cfConvertToFeaturelet()

cfConvertToFeaturelets()

```
#include <ch_cvl/edge.h>

cfConvertToFeaturelets();
```

Global function that converts a set of edgelets to an array of featurelets. Edgelets are described in the **ccEdgelet** and **ccEdgeletSet** reference pages and featurelets are described in the **ccFeaturelet** reference page.

cfConvertToFeaturelets

```
void cfConvertToFeaturelets (
  const ccEdgeletSet &edgeletSet,
  cmStd vector<ccFeaturelet> &featurelets);
```

Parameters

edgeletSet	The edgelet set to convert.
featurelets	The converted edgelets are placed here.

The following conversions are made for each edgelet:

Edgelet	Featurelet
position	*position* mapped by the edgelet set's *clientFromImageXform.*
angle	The function computes the edgelet tangent angle and maps that angle by the edgelet set's *clientFromImageXform.* The featurelet *angle* is then computed as the normal to this mapped edgelet angle.
magnitude	*magnitude*, same as edgelet.
	weight set to 1.
	isMod180 set to false.

Notes

The featurelets will be in the same order as the edgelets and correspond one-to-one with the edgelets in **edgelet.edges()**.

During initialization the *featurelets* array is cleared. Any featurelets provided on input are discarded.

If the *edgeletSet* is empty the *featurelets* array will be empty. The function does not throw an error for this case.

cfConvertToFeaturelets()

cfConvertToFeatures()

```
#include <ch_cvl/edge.h>
#include <ch_cvl/bndtrckr.h>
#include <ch_cvl/blobdesc.h>

cfConvertToFeatures();
```

This is a global function that converts vision tool results to featurelet chain sets. It has three overloads as follows:

1. Edge tool

   Converts a set of edgelets and an associated index chain list to a featurelets chain set. Edgelets are described in the **ccEdgelet** and **ccEdgeletSet** reference pages and index chain lists are described in **ccIndexChainList**.

2. Boundary Tracker tool

   Converts a boundary tracker result into a featurelet chain list. A boundary tracker result is described in the **ccBoundaryTrackerResult** reference page.

3. Blob tool

   Converts a blob scene description into a featurelet chain list. A blob scene description is described in the **ccBlobSceneDescription** reference page.

See **ccFeatureletChainSet** for information about featurelet chain sets.

Notes

The Edge tool overload is in *edge.h*, the Boundary Tracker tool overload is in *bndtrckr.h*, and the Blob tool overload is in *blobdesc.h*.

cfConvertToFeatures()

cfConvertToFeatures

```
ccFeatureletChainSetPtrh cfConvertToFeatures(
  const ccEdgeletSet& set,
  const ccIndexChainList& chains);

cmImport_cogip
ccFeatureletChainSetPtrh cfConvertToFeatures(
  ccBoundaryTrackerResult& result,
  bool decreasingAngleSideFirst = true,
  bool isWhiteOnBlack = false);

cmImport_cogblob
ccFeatureletChainSetPtrh cfConvertToFeatures(
  const ccBlobSceneDescription *bsd,
  bool isInverted = false,
  bool onlyTopLevel = false);
```

-
```
ccFeatureletChainSetPtrh cfConvertToFeatures(
  const ccEdgeletSet& set,
  const ccIndexChainList& chains);
```

Returns a featurelet chain set for the given edgelet set and index chain list.

Parameters

set	The edgelet set to convert.
chains	The index chain list for *edgeletSet*.

Throws

ccEdgeletDefs::BadParams

If the input chains have any out of bounds pointers to edgelets or edgelet indices.

There will be one chain in the returned featurelet chain set for each edgelet chain, and the featurelet chains will be in the same order as the edgelet chains.

A featurelet chain will be closed if and only if its corresponding edgelet chain is closed. Each featurelet chain will contain a featurelet for each edgelet in the corresponding edgelet chain, and in the same chain order.

The following conversions are made for each edgelet:

Edgelet	Featurelet
position	The edgelet *position* mapped by the edgelet set's *clientFromImageXform*
angle	The function computes the edgelet tangent angle by subtracting 90° from the edgelet's **angle()** and then mapping that tangent angle by the edgelet set's *clientFromImageXform*. The featurelet angle is then computed as the normal to this mapped tangent angle.
magnitude	Same as edgelet *magnitude*.
	weight set to 1.
	isMod180 set to false.

Notes

If the *edgeletSet* is empty the featurelets chain set will be empty. The function does not throw an error for this case.

-
```
cmImport_cogip
ccFeatureletChainSetPtrh cfConvertToFeatures(
  ccBoundaryTrackerResult& result,
  bool decreasingAngleSideFirst = true,
  bool isWhiteOnBlack = false);
```

Returns a featurelet chain set for the given boundary tracker result. The featurelet chain set will contain a single chain. The chain will contain a featurelet for each boundary tracker point, and in the same order. The chain will be open if, and only if, the *eOpenBoundary* status flag is set.

The *decreasingAngleSideFirst* and *isBlackOnWhite* parameters must be the same parameters used to generate the *result*. *isWhiteOnBlack* corresponds to the *eWhiteOnBlack* boundary tracker run-time parameter.

Parameters

result The boundary tracker result to be converted.

decreasingAngleSideFirst
The boundary tracker parameter used to generate *result*.

isWhiteOnBlack The boundary tracker parameter used to generate *result*.

The following conversions are made from boundary tracker results:

Boundary tracker	Featurelet
position	*position*, same as boundary tracker (client coordinates).
angle	*angle* normal to the boundary tracker angle and in the direction towards the white side of the shape. The featurelet angles are in the gradient direction (from negative to positive) normal to the **ccBoundaryTrackerPoint** angles which are in the tangent direction.
	magnitude is set to 0.
	weight set to 1.
	isMod180 set to false.

Notes

If no valid boundary has been tracked, or if the boundary tracker **statusFlags()** = 0, this function returns an empty featurelet chain set.

-
```
cmImport_cogblob
ccFeatureletChainSetPtrh cfConvertToFeatures(
  const ccBlobSceneDescription *bsd,
  bool isInverted = false,
  bool onlyTopLevel = false);
```

Returns a featurelet chain set for the given blob scene description. There is exactly one chain for each blob (except for non-top-level blobs which are ignored when the *onlyTopLevel* parameter is true). In each chain, there is exactly one featurelet for each *boundaryChainCode* element. The positions of the featurelets correspond to the points on the walk specified by the *boundaryChainCode* (transformed by the scene's clientFromImageXform). For example, if the blob scene description includes a single blob of a single pixel, then the featurelet chain set would include one featurelet chain, and the featurelet chain would include 4 featurelets, one at each corner of the pixel (corresponding to the boundary chain code which has four elements).

If *isInverted* is true, the featurelet chain set will correspond to a dark-on-bright region. If *onlyTopLevel* is true, the featurelet chain set only includes the top level blobs; otherwise, the featurelet chain set includes all of the blobs.

Parameters

bsd The blob scene description to convert.

isInverted Blob polarity inverted; true or false.

onlyTopLevel Convert all of the blobs, or just the top level; true or false.

The following conversions are made from blob scene description:

Blob	Featurelet
position	*position*, same as positions of *boundaryChainCode* elements, transformed by the blob scene's *clientFromImageXform*.
angle	The tangent angle at each point is computed in image coordinates as the average of the two incident vectors to the two adjacent points, and in the direction of the walk in the image. This tangent angle is then mapped by the scene's *clientFromImageXform*. The featurelet angle is computed as the normal of this mapped tangent angle, in client coordinates.
	magnitude is set to 0.
	weight set to 1.
	isMod180 set to false.

Notes

If the blob scene description is NULL, this function returns an empty featurelet chain set.

Because of the way blobs and featurelet chains are measured, measures such as area, center of mass, and bounding box computed from a featurelet chain set may be different than the same measures computed from the original blob scene description.

Boundaries corresponding to blobs with an odd depth (for example, the number of levels from the top-level blob) will have opposite handedness of the top-level blobs. Even depth blobs will have the same handedness of the top-level blobs.

Nominally, the featurelet chains corresponding to top-level blobs will be right-handed. If *isInverted* is true, the featurelet chains corresponding to top-level blobs will be left-handed.

cfConvertToFeatures()

cfConvertVDBtoCDB()

```
#include <ch_cvl/cdb.h>

cfConvertVDBtoCDB();
```

Global function that converts a CDB file to a VDB file.

cfConvertVDBtoCDB

```
cfConvertVDBtoCDB(const ccCvlString &vdbFile,
  const ccCvlString &cdbFile,
  ccCvlOStream *out = 0);
```

Converts VDB file named *vdbFile* to the CDB file named *cdbFile*.

Parameters

vdbFile	The name of the VDB file to convert.
cdbFile	The name of the CDB file to write.
out	An optional output stream to report errors and diagnostic messages.

Notes

Only records with acquired images are written into destination file.

cfConvertVDBtoCDB()

cfConvolve()

```
#include <ch_cvl/convolve.h>

cfConvolve();
```

Global function to convolve an image with a user-defined 3x3 kernel. You specify optional convolution parameters by supplying a **ccConvolveParams** object.

cfConvolve

```
template<class S, class D, class K> void cfConvolve(
  const ccPelBuffer_const<S>& src,
  const ccPelBuffer_const<K>& kernel, ccPelBuffer<D>& dst);

template<class S, class D, class K> void cfConvolve(
  const ccPelBuffer_const<S>& src,
  const ccPelBuffer_const<K>& kernel,
  const ccConvolveParams& params, ccPelBuffer<D>& dst);

template<class S, class K> ccPelBuffer<S> cfConvolve(
  const ccPelBuffer_const<S>& src,
  const ccPelBuffer_const<K>& kernel);

template<class S, class K> ccPelBuffer<S> cfConvolve(
  const ccPelBuffer_const<S>& src,
  const ccPelBuffer_const<K>& kernel,
  const ccConvolveParams& params);
```

- ```
  template<class S, class D, class K> void cfConvolve(
    const ccPelBuffer_const<S>& src,
    const ccPelBuffer_const<K>& kernel, ccPelBuffer<D>& dst);
  ```

  Convolves the supplied image with the supplied kernel using the default convolution parameters (automatic kernel prescaling and normalization and a kernel origin in the center of the kernel).

  Parameters

S	Template parameter specifying the type of the pixels in *src*. *S* must be **c_UInt8**.
D	Template parameter specifying the type of the pixels in *dst*. *D* must be **c_UInt8** or **c_Int16** and it must be the same as *K*.
K	Template parameter specifying the type of the pixels in *kernel*. *K* must be **c_UInt8** or **c_Int16** and it must be the same as *D*
src	The source image. *src* must be bound.
kernel	The kernel. *kernel* must be 3x3 in size and it must be bound.
  ```

dst The destination image.

If *dst* is bound, the only pixels in *src* that the tool considers are those that would produce convolved pixels lying within the greatest common region of *src* and *dst*.

If *dst* is unbound, a root image for *dst* is allocated all the pixels in *src* are convolved.

Throws

ccConvolveParams::NotImplemented
kernel is not 3x3 in size.

ccConvolveParams::UnboundWindow
src or *kernel* is unbound.

ccConvolveParams::BadArgument
The width or height of *src* is less than 3.

- ```
  template<class S, class D, class K> void cfConvolve(
    const ccPelBuffer_const<S>& src,
    const ccPelBuffer_const<K>& kernel,
    const ccConvolveParams& params, ccPelBuffer<D>& dst);
  ```

Convolves the supplied image with the supplied kernel using the specified convolution parameters.

Parameters

S Template parameter specifying the type of the pixels in *src*. *S* must be **c_UInt8**.

D Template parameter specifying the type of the pixels in *dst*. *D* must be **c_UInt8** or **c_Int16** and it must be the same as *K*.

K Template parameter specifying the type of the pixels in *kernel*. *K* must be **c_UInt8** or **c_Int16** and it must be the same as *D*

src The source image. *src* must be bound.

kernel The kernel. *kernel* must be 3x3 in size and it must be bound.

params A **ccConvolveParams** specifying the convolution parameters.

dst The destination image.

If *dst* is bound, the only pixels in *src* that the tool considers are those that would produce convolved pixels lying within the greatest common region of *src* and *dst*.
  ```

If *dst* is unbound, a root image for *dst* is allocated all the pixels in *src* are convolved.

Throws

ccConvolveParams::NotImplemented
kernel is not 3x3 in size.

ccConvolveParams::UnboundWindow
src or *kernel* is unbound.

ccConvolveParams::BadArgument
The width or height of *src* is less than 3; *params* specifies a negative normalization coefficient; or *params* specifies a kernel origin that does not lie within *kernel*.

- ```
  template<class S, class K> ccPelBuffer<S> cfConvolve(
    const ccPelBuffer_const<S>& src,
    const ccPelBuffer_const<K>& kernel);
  ```

Convolves the supplied image with the supplied kernel using the default convolution parameters (automatic kernel prescaling and normalization and a kernel origin in the center of the kernel) and returns the resulting convolved image.

Parameters

S Template parameter specifying the type of the pixels in *src*. *S* must be **c_UInt8**. The returned image will also be of type *S*.

K Template parameter specifying the type of the pixels in *kernel*. *K* must be **c_UInt8**.

src The source image. *src* must be bound.

kernel The kernel. *kernel* must be 3x3 in size and it must be bound.

Throws

ccConvolveParams::NotImplemented
kernel is not 3x3 in size.

ccConvolveParams::UnboundWindow
src or *kernel* is unbound.

ccConvolveParams::BadArgument
The width or height of *src* is less than 3.
  ```

- ```
  template<class S, class K> ccPelBuffer<S> cfConvolve(
    const ccPelBuffer_const<S>& src,
    const ccPelBuffer_const<K>& kernel,
    const ccConvolveParams& params);
  ```

  Convolves the supplied image with the supplied kernel using the specified convolution parameters and returns the resulting convolved image.

  Parameters

  S — Template parameter specifying the type of the pixels in *src*. *S* must be **c_UInt8**. The returned image will also be of type *S*.

  K — Template parameter specifying the type of the pixels in *kernel*. *K* must be **c_UInt8**.

  src — The source image. *src* must be bound.

  kernel — The kernel. *kernel* must be 3x3 in size and it must be bound.

  params — A **ccConvolveParams** specifying the convolution parameters.

  Throws

  ccConvolveParams::NotImplemented
  kernel is not 3x3 in size.

  ccConvolveParams::UnboundWindow
  src or *kernel* is unbound.

  ccConvolveParams::BadArgument
  The width or height of *src* is less than 3; *params* specifies a negative normalization coefficient; or *params* specifies a kernel origin that does not lie within *kernel*.
  ```

cfCreateThread()

```
#include <ch_cvl/threads.h>

cfCreateThread();
```

Global function to create a non-MFC thread.

Note This function is deprecated in CVL 6.0.1 and later.

In applications built with Microsoft Visual C++ .NET, you can no longer call MFC functions from non-MFC threads. Use the CVL global function **cfCreateThreadMFC()** to create an MFC thread, or **cfCreateThreadCVL()** to create a non-MFC thread.

For an overview of this subject, see the chapter Writing Multithreaded CVL Applications in the *CVL User's Guide*.

cfCreateThread()

cfCreateThreadCVL()

```
#include <ch_cvl/threads.h>

cfCreateThreadCVL();
```

Global function to create a non-MFC thread.

In applications built with Microsoft Visual C++.NET, you can no longer call MFC functions from non-MFC threads. Use **cfCreateThreadCVL()** rather than **cfCreateThread()**, which has been deprecated, to create a non-MFC thread. Use **cfCreateThreadMFC()** to create an MFC thread. For an overview of threads, see the chapter Writing Multithreaded CVL Applications in the *CVL User's Guide*.

cfCreateThreadCVL

```
ccThreadID cfCreateThreadCVL(
  void (* proc)(void *),
  void *arg,
  cePriority priority=cePriorityDefault,
  c_Int32 stackSize=-1);
```

Starts a new thread with the specified priority. Execution of the new thread begins with the supplied function.

Parameters

proc — The function with which to begin the thread. The supplied function must have the following signature:

```
void myThreadFunction(void *arg);
```

The *arg* argument will contain the value you supply when you call **cfCreateThreadCVL()**.

arg — The argument to be supplied to *proc* when it is invoked.

priority — The priority to assign to the new thread. *priority* must be one of the following values:

cePriorityDefault
cePriorityIdle
cePriorityLowest
cePriorityBelowNormal
cePriorityNormal
cePriorityAboveNormal
cePriorityHighest
cePriorityTimeCritical

If you specify *cePriorityDefault* (which is the default value) then the new thread is assigned the same priority as the current thread.

In CVL applications that will execute on the MVS-82400, do not create threads of priority *cePriorityTimeCritical*. If your application creates threads at this priority, your application on the embedded processor may experience problems with **ccTimer** objects and delays in image acquisition.

stackSize Size of the stack (specified in bytes) to reserve for this thread. A value of -1 means to use the operating system default. On the C80, the default stack size is 8K (8192 bytes).

Notes

Beginning with Visual C++ .NET, MFC will assert if you make MFC function calls from non-MFC threads. Always use MFC threads if a thread makes calls either to MFC functions or to CVL classes that use MFC (such as **ccDisplayConsole**). You can create MFC threads with either the CVL global function **cfCreateThreadMFC()** or the MFC function **AfxBeginThread()**.

Any errors thrown by the new thread are output to a **cogDebug** window before the thread terminates.

Throws

ccThreadID::CreateFailed
The thread could not be created.

cfCreateThreadMFC()

```
#include <ch_cvl/thrdsmfc.h>

cfCreateThreadMFC();
```

Global function to create an MFC worker thread. You should call MFC functions, and methods of objects that use MFC (such as **ccDisplayConsole**), only from MFC threads.

cfCreateThreadMFC

```
ccThreadID cfCreateThreadMFC(
  void (* proc)(void *),
  void *arg,
  cePriority priority=cePriorityDefault,
  c_Int32 stackSize=-1);
```

Creates a new MFC worker thread with the specified priority. The new thread starts execution with the supplied function **proc(arg)**.

Parameters

proc — The function with which to begin the thread. The supplied function must have the following signature:

```
void myFunctionName(void *arg);
```

arg — An argument that will be supplied to *proc* when it is invoked.

priority — The priority to assign to the new thread. Must be one of the following values:

cePriorityDefault
cePriorityIdle
cePriorityLowest
cePriorityBelowNormal
cePriorityNormal
cePriorityAboveNormal
cePriorityHighest
cePriorityTimeCritical

If the priority is *cePriorityDefault*, then the new thread is assigned the same priority as the calling thread.

In CVL applications that will execute on the MVS-82400, do not create threads of priority *cePriorityTimeCritical*. If your application creates threads at this priority, your application on the embedded processor may experience problems with **ccTimer** objects and delays in image acquisition.

stackSize Size of the stack (specified in bytes) to reserve for this thread. A value of -1 means to use the operating system default. On the C80, the default stack size is 8K (8192 bytes).

Notes

Beginning with Microsoft Visual C++ .NET, MFC will assert if you make MFC function calls from non-MFC threads. Always use MFC threads if a thread makes calls either to MFC functions or to CVL classes that use MFC (such as **ccDisplayConsole**). You can create MFC threads with either the CVL global function **cfCreateThreadMFC()** or the MFC function **AfxBeginThread()**.

Any errors thrown by the new thread are output to a **cogDebug** window before the thread terminates.

Throws

ccThreadID::CreateFailed
The MFC thread could not be created.

cfDefaultPelRootPool()

```
#include <ch_cvl/pelpool.h>

cfDefaultPelRootPool();
```

Global function to obtain the default root image pool (pel root pool).

Notes

This function does not apply to MVS-8600 and MVS-8600e frame grabbers. For these frame grabbers see **cc8600::sizePelRoot()**.

cfDefaultPelRootPool

```
const ccPelRootPoolPtrh& cfDefaultPelRootPool();

const ccPelRootPoolPtrh& cfDefaultPelRootPool(
  size_t size);
```

- `const ccPelRootPoolPtrh& cfDefaultPelRootPool();`

  Constructs a static pel root pool and returns a reference to it. The first time the function is called, the pel root pool is constructed. Later calls simply return a reference to the same pel root pool. Retrieve the size of the pool with **cfDefaultPelRootPoolSize** on page 3621.

- `const ccPelRootPoolPtrh& cfDefaultPelRootPool(size_t size);`

  Constructs a static pel root pool and returns a reference to it. The first time the function is called, the pel root pool is constructed. Later calls simply return a reference to the same pel root pool.

  Parameters

size	The size of the root image pool in bytes.

  Notes

  The default pel root pool size is determined the first time that **cfDefaultPelRootPool(size)** is called. If this function is not called directly, the first time it is called is normally the first time any FIFO is constructed. Acquisition FIFO construction fills the *pelRootPool* property with the return value from this function.

  You can change the default pel root pool size by calling **cfDefaultPelRootPool(size)** before constructing any FIFO. The return value can be dropped since it is the side effect of constructing the pel root pool that is important. Calling this function after a FIFO has been constructed will have no effect as the pel root pool will already have been constructed with the default 4MB size.

The default 4 MB pool can supply 9 full resolution CCIR buffers or 13 full resolution RS-170 pel roots. This may not be enough if buffers are held for some time instead of being recycled quickly.

Notes for the MVS-8100L:

The MVS-8100L normally uses two pel roots at a time when performing acquisitions. This is different than with other boards such as the MVS-8100M where only one pel root at a time is claimed for acquisition. This difference can show up as buffer starvation on the MVS-8100L in multiboard situations where the MVS-8100M does not show any problem.

cfDefaultPelRootPoolSize()

```
#include <ch_cvl/pelpool.h>

cfDefaultPelRootPoolSize();
```

Global function to obtain the default size of the default root image pool.

cfDefaultPelRootPoolSize

```
size_t cfDefaultPelRootPoolSize();
```

Returns the default size of the default root image pool in bytes. The default pel root pool size is 4 MB, unless the pool has been resized or the default size was overridden. You can retrieve the current size of the pool with **cfDefaultPelRootPool().size()**.

cfDefaultPelRootPoolSize()

cfDetectMouseBites()

```
#include <ch_cvl/pmifproc.h>

cfDetectMouseBites();
```

Global function to detect mousebites within a PatInspect feature.

cfDetectMouseBites

```
cmStd vector<ccMouseBite> cfDetectMouseBites(
  const ccPMInspectMatchedFeature& feature,
  c_Int32 minSize, double alpha);
```

Segments the supplied **cc_PMInspectFeature** into one or more **ccFeatureSegment**s.

Parameters

feature — A PatInspect feature.

minSize — Only mousebites larger than *minSize* are returned by the function. *minSize* is specified as the minimum number of feature boundary points in a mousebite.

alpha — The threshold value for mousebites. Only feature boundary points that have a distance from their expected position greater than *alpha* times the mean variation from the expected position of all points are considered as mousebites.

Note that if you run this function using the training image as the run-time image, the results may include unexpected mousebites. This happens because the feature boundary points are identical between the two images.

cfDetectMouseBites()

cfDetectSceneAngle()

```
#include <ch_cvl/scnangle.h>

cfDetectSceneAngle();
```

Global function to determine the angle of the predominant features in an image.

Notes

The first two overloads determine the scene angle using the Scene Angle Finder tool. The third overload determines the scene angle using the Scene Angle Finder-II tool.

cfDetectSceneAngle

```
void cfDetectSceneAngle (
  const ccPelBuffer_const<c_UInt8>& pelBufInputROI,
  const ccSceneAngleFinderRunParams& runParams,
  ccSceneAngleFinderResultSet& resultSet);

void cfDetectSceneAngle(
  const ccPelBuffer_const<c_UInt8>& pelBufInputROI,
  const ccEdgeletSet& edgeletSet,
  const ccSceneAngleFinderRunParams& runParams,
  ccSceneAngleFinderResultSet& resultSet);

void cfDetectSceneAngle(
  const ccPelBuffer_const<c_UInt8>& image,
   const ccSceneAngleFinderIIRunParams& runParams,
   ccSceneAngleFinderIIResultSet& resultSet);
```

- ```
  void cfDetectSceneAngle (
    const ccPelBuffer_const<c_UInt8>& pelBufInputROI,
    const ccSceneAngleFinderRunParams& runParams,
    ccSceneAngleFinderResultSet& resultSet);
  ```

  Computes and returns the angle of the predominant features within the supplied image using the Scene Angle Finder tool. The returned angle is the angle from the client coordinate x-axis to the positive gradient direction of the predominant features.

  Parameters

pelBufInputROI	The image whose scene angle is computed.
runParams	A **ccSceneAngleFinderRunParams** that contains the run-time parameters.
resultSet	A **ccSceneAngleFinderResultSet** into which the results are placed.
  ```

cfDetectSceneAngle()

- ```
  void cfDetectSceneAngle(
    const ccPelBuffer_const<c_UInt8>& pelBufInputROI,
    const ccEdgeletSet& edgeletSet,
    const ccSceneAngleFinderRunParams& runParams,
    ccSceneAngleFinderResultSet& resultSet);
  ```

  Computes and returns the angle of the predominant features described by the supplied **ccEdgeletSet** using the Scene Angle Finder tool. The returned angle is the angle from the client coordinate x-axis of the supplied image to the positive gradient direction of the predominant features in the **ccEdgeletSet** plus 90°.

  The contrast threshold, smoothing, and subsampling offset values in the supplied **ccSceneAngleFinderRunParams** are ignored by this function.

  Parameters

pelBufInputROI	An image whose client coordinate system should be the same as the client coordinate system associated with *edgeletSet*.
edgeletSet	A **ccEdgeletSet** that contains the subpixel edge information from which the scene angle is computed.
runParams	A **ccSceneAngleFinderRunParams** that contains the run-time parameters.
resultSet	A **ccSceneAngleFinderResultSet** into which the results are placed.

- ```
  void cmImport_cogip cfDetectSceneAngle(
    const ccPelBuffer_const<c_UInt8>& image,
     const ccSceneAngleFinderIIRunParams& runParams,
     ccSceneAngleFinderIIResultSet& resultSet);
  ```

  Computes and returns the angle of the predominant features within the supplied image using the Scene Angle Finder-II tool. The returned angle is the angle from the client coordinate x-axis to the positive gradient direction of the predominant features plus 90°.

  Parameters

image	The image whose scene angle is computed.
runParams	A **ccSceneAngleFinderIIRunParams** that contains the run-time parameters.
resultSet	A **ccSceneAngleFinderIIResultSet** into which the results are placed.

  Notes

  image must be at least 32x32 pixels in size.

cfDrawSynthetic()

```
#include <ch_cvl/genpoly.h>

cfDrawSynthetic();
```

Note This function is deprecated. Use **cfRasterize()** instead.

Global function to draw a synthetic version of the supplied closed shape in the supplied pel buffer. The clientFromShape transform specifies how this shape maps into the client coordinates of the pel buffer. The inside of the shape is rendered in the supplied grey value. Pixels that fall atop any segment are rendered in an interpolated grey level that is between the inside grey level and the background grey level in the area immediately outside the generalized polygon at that pixel.

cfDrawSynthetic

```
void cfDrawSynthetic(const ccGenPoly& shape,
  const cc2Xform& clientFromShape,
  ccPelBuffer<c_UInt8>& pelbuf,
  c_UInt8> insideGrayLevel);

void cfDrawSynthetic(const ccGenPoly& shape,
  const cc2XformBasePtrh_const& clientFromShape,
  ccPelBuffer<c_UInt8>& pelbuf,
  c_UInt8 insideGrayLevel);
```

- ```
  void cfDrawSynthetic(const ccGenPoly& shape,
    const cc2Xform& clientFromShape,
    ccPelBuffer<c_UInt8>& pelbuf,
    c_UInt8> insideGrayLevel);
  ```

  Draws a synthetic version of the supplied closed shape in the supplied pel buffer.

  Parameters

  shape The closed shape

  clientFromShape Transform that maps the shape into the client coordinates of the pel buffer. The transform is provided as a two-dimensional transformation object (**cc2Xform**).

  pelbuf The pel buffer in which the synthetic shape is to be drawn

  insideGrayLevel The pixel value of the area inside the border of the closed shape

  Throws

  ccShapesError::BadGeom
  shape is not closed
  ```

- ```
  void cfDrawSynthetic(const ccGenPoly& shape,
    const cc2XformBasePtrh_const& clientFromShape,
    ccPelBuffer<c_UInt8>& pelbuf,
    c_UInt8 insideGrayLevel);
  ```

  Draws a synthetic version of the supplied closed shape in the supplied pel buffer.

  Parameters

  shape The closed shape

  clientFromShape Transform that maps the shape into the client coordinates of the pel buffer. The transform is provided as a constant pointer handle to a generic two-dimensional transformation object (**cc2XformBasePtrh_const**).

  pelbuf The pel buffer in which the synthetic shape is to be drawn

  insideGrayLevel The pixel value of the area inside the border of the closed shape

  Throws

  ccShapesError::BadGeom
  shape is not closed
  ```

cfEdgeDetect()

```
#include <ch_cvl/edge.h>

cfEdgeDetect();
```

Global function to apply the Edge tool to an input image.

cfEdgeDetect

```
template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params, ccEdgeletSet& dst,
  ccDiagObject* obj, c_UInt32 diagflags);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
```

```
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagflags);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccDiagObject* obj);

template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params, ccEdgeletSet& dst);

template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params, ccEdgeletSet& dst
  ccDiagObject* obj);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccDiagObject* obj);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccIndexChainList& chains);

template<class S, class D>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
```

```
  ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj);

template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains);

template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<c_UInt8>& dstMag,
  ccPelBuffer<c_UInt8>& dstAng,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<c_UInt8>& dstMag,
  ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<c_UInt8>& dstMag,
```

```
  ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<c_UInt8>& dstMag,
  ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<c_UInt8>& dstMag,
  ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
  const ccEdgeletParams& params,
  ccPelBuffer<c_UInt8>& dstMag,
```

```
  ccPelBuffer<c_UInt8>& dstAng,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);

template<>
void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);
```

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Edge magnitude and edge angle images are written to the supplied pel buffers. Diagnostic information is written to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**.
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

- ```
  template<class S>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params, ccEdgeletSet& dst,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dst	The edgelet set in which to store the edgelet objects.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Edge magnitude and edge angle images are written to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
  ```

```
ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
ccEdgeletSet& dstSet, ccIndexChainList& chains,
ccDiagObject* obj, c_UInt32 diagFlags);
```

Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

-
```
template<class S>
void cfEdgeDetect(const ccPelBuffer_const<S>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);
```

Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dstSet	The edgelet set in which the edgelet objects are stored.

chains Index chain list ordering the edgelets in the edgelet set.

obj The diagnostic object in which to store diagnostic information.

diagFlags The diagnostic flags (see **ccDiagDefs**).

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Edge magnitude and edge angle images are stored to the supplied pel buffers. Creates an empty diagnostic object with diagnostic flags set to zero.

  Parameters

  src The input image. Must be of type **c_UInt8** or **c_Int16**

  params The edgelet parameters used for edge detection.

  dstMag The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.

  dstAng The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccDiagObject* obj);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Stores diagnostic information to the supplied diagnostic object with diagnostic flags set to zero.

  Parameters

  src The input image. Must be of type **c_UInt8** or **c_Int16**

  params The edgelet parameters used for edge detection.

  dstMag The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.

dstAng The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

obj The diagnostic object in which to store diagnostic information.

- ```
  template<class S>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccEdgeletSet& dst);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Creates a empty diagnostic object with diagnostic flags set to zero.

  Parameters

  src The input image. Must be of type **c_UInt8** or **c_Int16**

  params The edgelet parameters used for edge detection.

  dst The edgelet set in which the edgelet objects are stored.

- ```
  template<class S>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccEdgeletSet& dst, ccDiagObject* obj);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object with diagnostic flags set to zero.

  Parameters

  src The input image. Must be of type **c_UInt8** or **c_Int16**

  params The edgelet parameters used for edge detection.

  dst The edgelet set in which the edgelet objects are stored.

  obj The diagnostic object in which to store diagnostic information.

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Creates an empty diagnostic object with diagnostic flags set to zero.

  Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet, ccDiagObject* obj);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object with diagnostic flags set to zero.

  Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

dstSet The edgelet set in which the edgelet objects are stored.

obj The diagnostic object in which to store diagnostic information.

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet, ccIndexChainList& chains);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Creates an empty diagnostic object with diagnostic flags set to zero.

  Parameters

  src The input image. Must be of type **c_UInt8** or **c_Int16**

  params The edgelet parameters used for edge detection.

  dstMag The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.

  dstAng The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

  dstSet The edgelet set in which the edgelet objects are stored.

  chains Index chain list ordering the edgelets in the edgelet set.

- ```
  template<class S, class D>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<D>& dstMag, ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet, ccIndexChainList& chains,
    ccDiagObject* obj);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object with diagnostic flags set to zero.

  Parameters

  src The input image. Must be of type **c_UInt8** or **c_Int16**

params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound. *dstMag* must be of type **c_UInt8**.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.
obj	The diagnostic object in which to store diagnostic information.

- ```
  template<class S>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccEdgeletSet& dstSet, ccIndexChainList& chains);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Creates an empty diagnostic object with diagnostic flags set to zero.

  Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**
params	The edgelet parameters used for edge detection.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.

- ```
  template<class S>
  void cfEdgeDetect(const ccPelBuffer_const<S>& src,
    const ccEdgeletParams& params,
    ccEdgeletSet& dstSet, ccIndexChainList& chains,
    ccDiagObject* obj);
  ```

  Applies the Edge tool to the supplied input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object with diagnostic flags set to zero.

  Parameters

src	The input image. Must be of type **c_UInt8** or **c_Int16**

params The edgelet parameters used for edge detection.

dstSet The edgelet set in which the edgelet objects are stored.

chains Index chain list ordering the edgelets in the edgelet set.

obj The diagnostic object in which to store diagnostic information.

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<c_UInt8>& dstMag,
    ccPelBuffer<c_UInt8>& dstAng,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 8-bit input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

  src The 8-bit input image.

  params The edgelet parameters used for edge detection.

  dstMag The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound.

  dstAng The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

  obj The diagnostic object in which to store diagnostic information.

  diagFlags The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
    const ccEdgeletParams& params,
    ccEdgeletSet& dstSet,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 8-bit input image using the supplied edgelet parameters. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

  src The 8-bit input image.

  params The edgelet parameters used for edge detection.

dstSet	The edgelet set in which the edgelet objects are stored.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<c_UInt8>& dstMag,
    ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 8-bit input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

src	The 8-bit input image.
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<c_UInt8>& dstMag,
  ```

```
ccPelBuffer<c_UInt8>& dstAng,
ccEdgeletSet& dstSet, ccIndexChainList& chains,
ccDiagObject* obj, c_UInt32 diagFlags);
```

Applies the Edge tool to the supplied 8-bit input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The 8-bit input image.
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

-
```
template<>
void cfEdgeDetect(const ccPelBuffer_const<c_UInt8>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);
```

Applies the Edge tool to the supplied 8-bit input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The 8-bit input image.
params	The edgelet parameters used for edge detection.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.

obj The diagnostic object in which to store diagnostic information.

diagFlags The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<c_UInt8>& dstMag,
    ccPelBuffer<c_UInt8>& dstAng,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 16-bit input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

  src The 16-bit input image.

  params The edgelet parameters used for edge detection.

  dstMag The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound.

  dstAng The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

  obj The diagnostic object in which to store diagnostic information.

  diagFlags The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
    const ccEdgeletParams& params,
    ccEdgeletSet& dstSet,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 16-bit input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

  src The 16-bit input image.

  params The edgelet parameters used for edge detection.

  dstSet The edgelet set in which the edgelet objects are stored.

obj The diagnostic object in which to store diagnostic information.

diagFlags The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<c_UInt8>& dstMag,
    ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 16-bit input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

  Parameters

  src The 16-bit input image.

  params The edgelet parameters used for edge detection.

  dstMag The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound.

  dstAng The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.

  dstSet The edgelet set in which the edgelet objects are stored.

  obj The diagnostic object in which to store diagnostic information.

  diagFlags The diagnostic flags (see **ccDiagDefs**).

- ```
  template<>
  void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
    const ccEdgeletParams& params,
    ccPelBuffer<c_UInt8>& dstMag,
    ccPelBuffer<c_UInt8>& dstAng,
    ccEdgeletSet& dstSet, ccIndexChainList& chains,
    ccDiagObject* obj, c_UInt32 diagFlags);
  ```

  Applies the Edge tool to the supplied 16-bit input image using the supplied edgelet parameters. Stores edge magnitude and edge angle images to the supplied pel buffers. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The 16-bit input image.
params	The edgelet parameters used for edge detection.
dstMag	The pel buffer in which to store the edge magnitude image. If *dstMag* is unbound, a root image is allocated and bound.
dstAng	The pel buffer in which to store the edge angle image. If *dstAng* is unbound, a root image is allocated and bound.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

-
```
template<>
void cfEdgeDetect(const ccPelBuffer_const<c_Int16>& src,
  const ccEdgeletParams& params,
  ccEdgeletSet& dstSet, ccIndexChainList& chains,
  ccDiagObject* obj, c_UInt32 diagFlags);
```

Applies the Edge tool to the supplied 16-bit input image using the supplied edgelet parameters. Returns an edgelet set containing an edgelet object for each edge in the input image. Returns an index chain list ordering the edgelets in the edgelet set. Stores diagnostic information to the supplied diagnostic object using the supplied diagnostic flags.

Parameters

src	The 16-bit input image.
params	The edgelet parameters used for edge detection.
dstSet	The edgelet set in which the edgelet objects are stored.
chains	Index chain list ordering the edgelets in the edgelet set.
obj	The diagnostic object in which to store diagnostic information.
diagFlags	The diagnostic flags (see **ccDiagDefs**).

cfEllipseFit()

```
#include <ch_cvl/fit.h>

cfEllipseFit();
```

Global function to fit an ellipse to a set of points.

cfEllipseFit

```
void cfEllipseFit(
  const ccEllipseFitParams &params,
  cmStd vector<cc2Vect> &pts,
  ccEllipseFitResults &results);
```

A global function that fits an Ellipse to the supplied set of points using the supplied **ccEllipseFitParams**.

Parameters

params — A **ccEllipseFitParams** describing the fitting method, number of outliers to ignore, and maximum RMS error threshold.

pts — A vector of points specified in a single coordinate system.

results — A reference to a **ccEllipseFitResults** object into which the results will be placed. Any existing contents of *results* are cleared.

Throws

ccMathError::Singular

If **params.numIgnore()** > **pts.size()** - 5,
or if an ellipse cannot be fitted to the given data points (*pts*) using the given parameters.

cfEllipseFit()

cfEqualize_1()

```
#include <ch_cog/equalize.h>

cfEqualize_1();
```

Global function to equalize brightness and contrast on a dual-tap **ccAcqFifo**.

cfEqualize_1

```
bool cfEqualize_1(ccGreyAcqFifoPtrh fifo);
```

This function automatically acquires a series of images from the supplied **ccAcqFifo** and adjusts the brightness and contrast values on the two channels (starting with the odd channel) until they are balanced.

The residual difference in grey level values between the fields after calling this function depends on the initial contrast value that is specified for the supplied **ccAcqFifo**. The following table shows the maximum residual grey level difference between the fields after calling this function with different initial contrast levels.

Contrast	Maximum Residual Difference
0.0	± 2 grey levels
0.5	± 4 grey levels
0.75	± 8 grey levels

Note that if you call **cfEqualize_1()** on a **ccAcqFifo** that is already balanced within the limits shown in the preceding table, the function may *increase* the imbalance, although the residual imbalance will remain within the limits shown in the preceding table.

This function returns true if the supplied **ccAcqFifo** was balanced. It returns false if it was unable to balance the fields due to an internal error.

Parameters

fifo The **ccAcqFifo** to equalize.

Notes

You must include the file *ch_cog/equalize.h* in your source file to use this function.

This function saves and restores the current input lookup table associated with *fifo*. The equalization is performed using the *positive8BitLut* linear 8-bit lookup table.

If you change the contrast or brightness of the **ccAcqFifo** after calling this function, the field balance may be lost. You should always call **cfEqualize_1()** after adjusting the **ccAcqFifo**'s brightness or contrast.

cfEqualize_1()

cfFilterConvolve()

```
#include <ch_cvl/filtconv.h>

cfFilterConvolve();
```

Templated global function to perform discrete convolution using a variable-sized kernel.

cfFilterConvolve

```
void cfFilterConvolve(const ccPelBuffer_const<S>& src,
     const ccFilterConvolveParams& params,
     const ccFilterConvolveKernel<K>& kernel,
     ccPelBuffer<D>& dst);

ccPelBuffer<K> cfFilterConvolve(
     const ccPelBuffer_const<S>& src,
     const ccFilterConvolveParams& params,
     const ccFilterConvolveKernel<K>& kernel);
```

- ```
  void cfFilterConvolve(const ccPelBuffer_const<S>& src,
       const ccFilterConvolveParams& params,
       const ccFilterConvolveKernel<K>& kernel,
       ccPelBuffer<D>& dst);
  ```

  Convolves the source image using the supplied kernel and convolution parameters object.

  Parameters

src	The image to convolve. This must be of type **c_UInt8**.
params	The convolution parameters to use.
kernel	The convolution kernel. This may be of type **c_UInt8** or **c_UInt16** and it must be of the same type as *dst*.
dst	The destination image. This may be of type **c_UInt8** or **c_UInt16** and it must be of the same type as *kernel*.

- ```
  ccPelBuffer<K> cfFilterConvolve(
       const ccPelBuffer_const<S>& src,
       const ccFilterConvolveParams& params,
       const ccFilterConvolveKernel<K>& kernel);
  ```

  Convolves the source image using the supplied kernel and convolution parameters object. This overload returns the convolved image.

  Parameters

src	The image to convolve. This must be of type **c_UInt8**.

params	The convolution parameters to use.
kernel	The convolution kernel. This may be of type **c_UInt8** or **c_UInt16** and it must be of the same type as *dst*.

cfFilterEdgeletChains()

```
#include <ch_cvl/edgefilt.h>

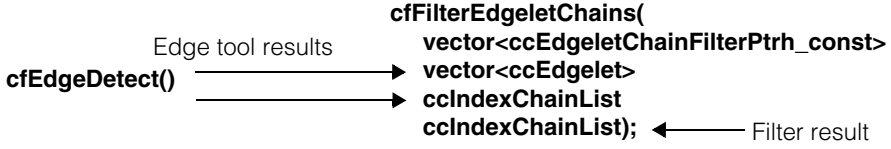
cfFilterEdgeletChains();
```

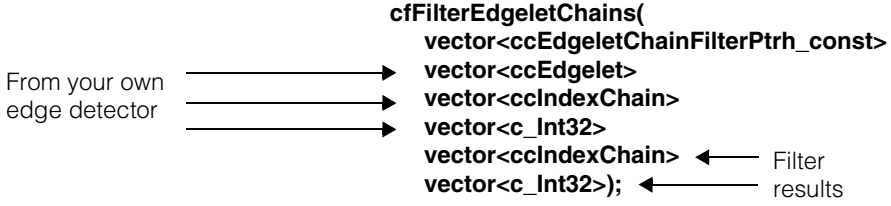
Global function to run the Edgelet Chain Filter tool.

An edgelet is a data structure that defines an edge at a sub-pixel location along with its angle and magnitude. Vision tools such as the Edge tool find edgelets in images and store them in a result array. The Edge tool also produces an edgelet chain list which is an associated array describing groups (chains) of related edgelets.

After running an Edge tool and acquiring the raw edgelets, you may wish to run a filter on the edgelets to eliminate certain edgelets and keep only the edgelets most pertinent to your application. You call **cfFilterEdgeletChains()** to do this filtering.

cfFilterEdgeletChains() has two overloads as shown below. The first overload is designed to work with Cognex Edge tool results as shown. The second overload is a general purpose filter intended to be used with an edge detector you design.





cfFilterEdgeletChains()

cfFilterEdgeletChains

```
void cfFilterEdgeletChains(
  const cmStd vector<ccEdgeletChainFilterPtrh_const>
                                                  &filters,
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const ccIndexChainList &inputChains,
  ccIndexChainList &filteredChains);

void cfFilterEdgeletChains(
  const cmStd vector<ccEdgeletChainFilterPtrh_const>
                                                  &filters,
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const cmStd vector<ccIndexChain> &inputChains,
  const cmStd vector<c_Int32> &inputIndexVector,
  cmStd vector<ccIndexChain> &filteredChains,
  cmStd vector<c_Int32> &filteredIndexVector);
```

This function performs filtering on the provided edgelet chains. The filtering is performed by calling the virtual member function **filter()** of each routine in *filters*. The filter at vector index 0 is called first, and subsequent filters are called in their vector position order. The resulting chains from running each filter are fed to the next filter in the vector. The final result is the result of the last filter.

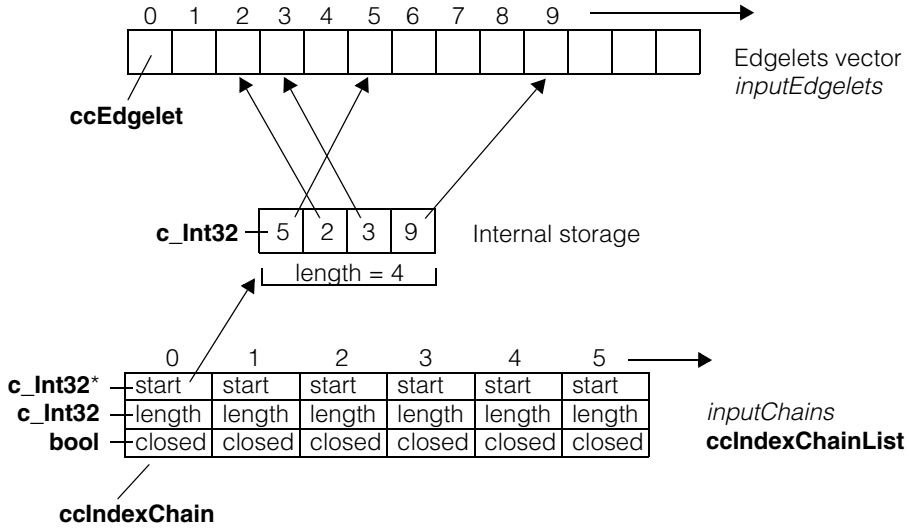
The filtering will be performed correctly even if *inputChains* and *filteredChains* are references to the same vector.

Any existing elements in *filteredChains* will be discarded before the filtering starts.

-
```
void cfFilterEdgeletChains(
  const cmStd vector<ccEdgeletChainFilterPtrh_const>
                                                  &filters,
  const cmStd vector<ccEdgelet> &inputEdgelets,
  const ccIndexChainList &inputChains,
  ccIndexChainList &filteredChains);
```

This overload is intended to be used with the Cognex Edge tool. The *inputEdgelets* and *inputChains* parameters are Edge tool outputs.

The following simple example shows how the edgelets are linked together into chains. In this example, the first chain is an open chain of four edgelets.



The filtered output is simply a reconfigured *inputChains* contained in *filteredChains*. The *inputEdgelets* vector is not changed.

Parameters

filters	A vector of filter routines you wish to run on the edgelets. The filters are run in the order they appear in the vector.
inputEdgelets	The edgelets to be filtered.
inputChains	The input edgelet chains used with the first filter. All filters except the first filter use the previous filter result (in *filteredChains*) as the input chain.
filteredChains	The filtered results.

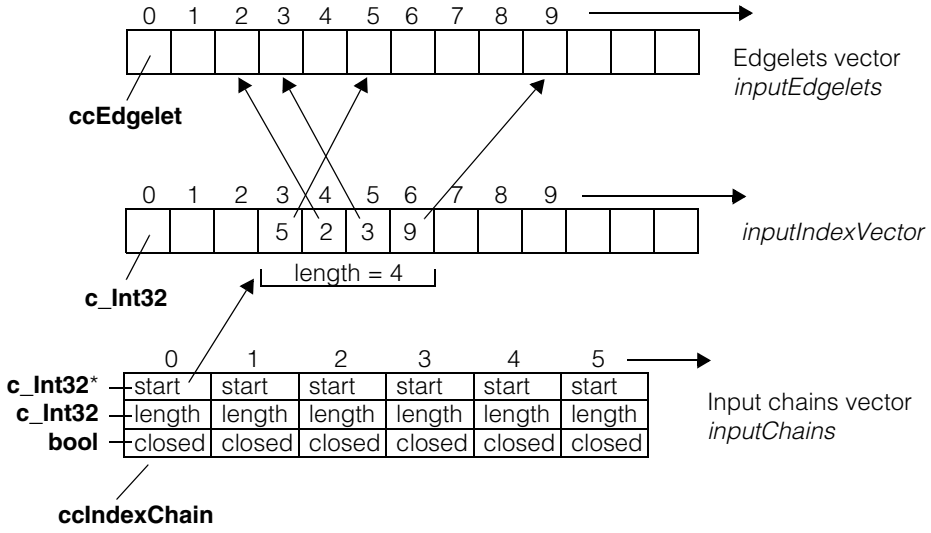
Throws

This function does not mask any throws from the filter routines.

- ```
  void cfFilterEdgeletChains(
    const cmStd vector<ccEdgeletChainFilterPtrh_const>
                                                    &filters,
    const cmStd vector<ccEdgelet> &inputEdgelets,
    const cmStd vector<ccIndexChain> &inputChains,
    const cmStd vector<c_Int32> &inputIndexVector,
    cmStd vector<ccIndexChain> &filteredChains,
    cmStd vector<c_Int32> &filteredIndexVector);
  ```

  This overload is intended for users that do not use the Cognex Edge tool. The edgelet chaining is more general purpose and may be easier to use when you are creating your own edgelet chains.

  The following simple example shows how the edgelets are linked together into chains. In this example, the first chain is an open chain of four edgelets. The *inputIndexVector* contains indices into the *inputEdgelets* vector.



  The filtered output is simply a reconfigured *inputChains* contained in *filteredChains*, and a reconfigured *inputIndexVector* contained in *filteredIndexVector*. The *inputEdgelets* vector is not changed.

  Parameters

filters	A vector of filter routines you wish to run on the edgelets. The filters are run in the order they appear in the vector.
inputEdgelets	The edgelets to be filtered.
inputChains	The input edgelet chains.
  ```

inputIndexVector The input indices into the edgelets vector.

filteredChains The filtered chains.

filteredIndexVector
The filtered indices into the edgelets vector.

Throws

This function does not mask any throws from the filter routines.

cfFilterEdgeletChains()

cfFilterMedian()

```
#include <ch_cvl/filtmed.h>

cfFilterMedian();
```

Global function to perform a median filter using a variable-sized kernel.

cfFilterMedian

```
void cfFilterMedian(const ccPelBuffer_const<c_UInt8>& src,
     const ccFilterMedianParams& params,
     const ccFilterMaskKernel& kernel,
     ccPelBuffer<c_UInt8>& dst);

void cfFilterMedian(const ccPelBuffer_const<c_UInt8>& src,
     const ccPelBuffer_const<c_UInt8>& mask,
     const ccFilterMedianParams& params,
     const ccFilterMaskKernel& kernel,
     ccPelBuffer<c_UInt8>& dst);

ccPelBuffer<c_UInt8> cfFilterMedian(
     const ccPelBuffer_const<c_UInt8>& src,
     const ccFilterMedianParams& params,
     const ccFilterMaskKernel& kernel);

ccPelBuffer<c_UInt8> cfFilterMedian(
     const ccPelBuffer_const<c_UInt8>& src,
     const ccPelBuffer_const<c_UInt8>& mask,
     const ccFilterMedianParams& params,
     const ccFilterMaskKernel& kernel);
```

-
```
void cfFilterMedian(const ccPelBuffer_const<c_UInt8>& src,
     const ccFilterMedianParams& params,
     const ccFilterMaskKernel& kernel,
     ccPelBuffer<c_UInt8>& dst);
```

Filters the supplied image using the supplied kernel and places the result in the supplied reference.

Parameters

src	The image to process.
params	A **ccFilterMedianParams** object specifying the filtering parameters.
kernel	The kernel with which to filter the image.
dst	A reference into which to place the filtered image.

cfFilterMedian ()

- ```
  void cfFilterMedian(const ccPelBuffer_const<c_UInt8>& src,
      const ccPelBuffer_const<c_UInt8>& mask,
      const ccFilterMedianParams& params,
      const ccFilterMaskKernel& kernel,
      ccPelBuffer<c_UInt8>& dst);
  ```

  Filters the supplied image using the supplied kernel and places the result in the supplied reference. Only image pixels that correspond to care pixels (255) in the supplied mask are processed.

  Parameters

src	The image to process.
params	A **ccFilterMedianParams** object specifying the filtering parameters.
mask	The mask image.
kernel	The kernel with which to filter the image.
dst	A reference into which to place the filtered image.

- ```
  ccPelBuffer<c_UInt8> cfFilterMedian(
      const ccPelBuffer_const<c_UInt8>& src,
      const ccFilterMedianParams& params,
      const ccFilterMaskKernel& kernel);
  ```

  Filters the supplied image using the supplied kernel and returns the result.

  Parameters

src	The image to process.
params	A **ccFilterMedianParams** object specifying the filtering parameters.
kernel	The kernel with which to filter the image.

- ```
  ccPelBuffer<c_UInt8> cfFilterMedian(
      const ccPelBuffer_const<c_UInt8>& src,
      const ccPelBuffer_const<c_UInt8>& mask,
      const ccFilterMedianParams& params,
      const ccFilterMaskKernel& kernel);
  ```

  Filters the supplied image using the supplied kernel and returns the result. Only image pixels that correspond to care pixels (255) in the supplied mask are processed.

  Parameters

src	The image to process.
  ```

params	A **ccFilterMedianParams** object specifying the filtering parameters.
mask	The mask image.
kernel	The kernel with which to filter the image.

cfFilterMedian ()

cfFilterMorphology()

```
#include <ch_cvl/filtmorph.h>

cfFilterMorphology();
```

Global function to perform grey-scale morphology using a variable-sized kernel.

cfFilterMorphology

```
void cfFilterMorphology(
     const ccPelBuffer_const<c_UInt8>& src,
     ccFilterMorphologyDefs::Operation operation,
     const ccFilterMorphologyParams& params,
     const ccFilterMorphologyStructuringElement& kernel,
     ccPelBuffer<c_UInt8>& dst);

void cfFilterMorphology(
     const ccPelBuffer_const<c_UInt8>& src,
     const ccPelBuffer_const<c_UInt8>& mask,
     ccFilterMorphologyDefs::Operation operation, const
     ccFilterMorphologyParams& params, const
     ccFilterMorphologyStructuringElement& kernel,
     ccPelBuffer<c_UInt8>& dst);

ccPelBuffer<c_UInt8> cfFilterMorphology(
     const ccPelBuffer_const<c_UInt8>& src,
     ccFilterMorphologyDefs::Operation operation, const
     ccFilterMorphologyParams& params, const
     ccFilterMorphologyStructuringElement& kernel);

ccPelBuffer<c_UInt8> cfFilterMorphology(
     const ccPelBuffer_const<c_UInt8>& src,
     const ccPelBuffer_const<c_UInt8>& mask,
     ccFilterMorphologyDefs::Operation operation, const
     ccFilterMorphologyParams& params, const
     ccFilterMorphologyStructuringElement& kernel);
```

- ```
  void cfFilterMorphology(
       const ccPelBuffer_const<c_UInt8>& src,
       ccFilterMorphologyDefs::Operation operation, const
       ccFilterMorphologyParams& params, const
       ccFilterMorphologyStructuringElement& kernel,
       ccPelBuffer<c_UInt8>& dst);
  ```

  Applies the specified morphological operator to the supplied source image using the supplied structuring element and places the result in the supplied reference.

  Parameters

  src The image to process.
  ```

operation	The morphological operation. Must be one of *ccFilterMorphologyDefs:::eDilate* *ccFilterMorphologyDefs:::eErode* *ccFilterMorphologyDefs:::eOpen* *ccFilterMorphologyDefs:::eClose*
params	A **ccFilterMorphologyParams** object specifying the filtering parameters.
kernel	The structuring element.
dst	A reference into which to place the filtered image.

- ```
  void cfFilterMorphology(
      const ccPelBuffer_const<c_UInt8>& src, const
      ccPelBuffer_const<c_UInt8>& mask,
      ccFilterMorphologyDefs::Operation operation, const
      ccFilterMorphologyParams& params, const
      ccFilterMorphologyStructuringElement& kernel,
      ccPelBuffer<c_UInt8>& dst);
  ```

  Applies the specified morphological operator to the supplied source image using the supplied structuring element and places the result in the supplied reference. Only image pixels that correspond to care pixels (255) in the supplied mask are processed.

  Parameters

src	The image to process.
mask	The mask image.
operation	The morphological operation. Must be one of *ccFilterMorphologyDefs:::eDilate* *ccFilterMorphologyDefs:::eErode* *ccFilterMorphologyDefs:::eOpen* *ccFilterMorphologyDefs:::eClose*
params	A **ccFilterMorphologyParams** object specifying the filtering parameters.
kernel	The structuring element.
dst	A reference into which to place the filtered image.
  ```

- ```
  ccPelBuffer<c_UInt8> cfFilterMorphology(
      const ccPelBuffer_const<c_UInt8>& src,
      ccFilterMorphologyDefs::Operation operation, const
      ccFilterMorphologyParams& params, const
      ccFilterMorphologyStructuringElement& kernel);
  ```

  Applies the specified morphological operator to the supplied source image using the supplied structuring element returns the result.

  Parameters

src	The image to process.
operation	The morphological operation. Must be one of *ccFilterMorphologyDefs:::eDilate* *ccFilterMorphologyDefs:::eErode* *ccFilterMorphologyDefs:::eOpen* *ccFilterMorphologyDefs:::eClose*
params	A **ccFilterMorphologyParams** object specifying the filtering parameters.
kernel	The structuring element.

- ```
  ccPelBuffer<c_UInt8> cfFilterMorphology(
      const ccPelBuffer_const<c_UInt8>& src, const
      ccPelBuffer_const<c_UInt8>& mask,
      ccFilterMorphologyDefs::Operation operation, const
      ccFilterMorphologyParams& params, const
      ccFilterMorphologyStructuringElement& kernel);
  ```

  Applies the specified morphological operator to the supplied source image using the supplied structuring element and returns the result. Only image pixels that correspond to care pixels (255) in the supplied mask are processed.

  Parameters

src	The image to process.
mask	The mask image.
operation	The morphological operation. Must be one of *ccFilterMorphologyDefs:::eDilate* *ccFilterMorphologyDefs:::eErode* *ccFilterMorphologyDefs:::eOpen* *ccFilterMorphologyDefs:::eClose*
params	A **ccFilterMorphologyParams** object specifying the filtering parameters.

kernel The structuring element.

Notes

If the structuring element has *useOffsetImage* set to false, then its *offsetImage* (if any) is ignored.

Refer to <*ch_cvl/filtcomm.h*> for more detailed descriptions shared by all NxM image processing filters.

cfFreeRunTrigger()

```
#include <ch_cvl/trigmodl.h>

cfFreeRunTrigger();
```

Global function to retrieve a free run trigger model object reference.

cfFreeRunTrigger

```
const ccTriggerModel& cfFreeRunTrigger();
```

When you specify **cfFreeRunTrigger()** as the trigger model, the software attempts to acquire at the maximum rate possible from the connected camera. The application should not call **ccAcqFifo::start()** to initiate image acquisitions.

If you call **ccAcqFifo::start()** when the free run trigger model is selected, your application throws *ccAcqFifo::StartNotAllowed*.

When **triggerEnable()** is false, no acquistions are performed. To allow the acquisitions to continue, **triggerEnable()** must be set to true.

Notes

If you change property values while **ccTriggerProp::triggerEnable()** is set to true, the changes will take place after an unspecified number of acquisitions. To force changes in property values to take effect with the next acquisition, set **triggerEnable()** to false, change the properties, then set **triggerEnable()** to true. Alternatively, you may want to consider using the semi trigger model.

Throws

ccAcqFifo::StartNotAllowed

You called **ccAcqFifo::start()**, which is not supported and not allowed with **cfFreeRunTrigger()**.

cfFreeRunTrigger()

cfGaussSample()

```
#include <ch_cvl/gaussmpl.h>

cfGaussSample();
```

Global function to perform Gaussian smoothing and sampling of an input image.

cfGaussSample

```
template<class T>
ccPelBuffer<T> cfGaussSample(
  const ccPelBuffer_const<T>& src,
  const ccGaussSampleParams& params);

template<class T>
void cfGaussSample(
  const ccPelBuffer_const<T>& src, ccPelBuffer<T>& dst,
  const ccGaussSampleParams& params);

template<class T>
void cfGaussSample(
  const ccPelBuffer_const<T>& src, ccPelBuffer<U>& dst,
  const ccGaussSampleParams& params);
```

- ```
  template<class T>
  ccPelBuffer<T> cfGaussSample(
    const ccPelBuffer_const<T>& src,
    const ccGaussSampleParams& params);
  ```

  Computes and returns an image by applying a Gaussian filter to the supplied input image using the supplied **ccGaussSampleParams**.

  This overload is instantiated for 8-bit input and output pel buffers.

  Parameters

  src — The input image. It must be of type **c_UInt8**.

  params — A **ccGaussSampleParams** specifying the sampling and smoothing parameters.

  Throws

  ccGaussSampleParams::BadParams
  The sampling rate specified in *params* is such that an output image with a height or width of zero would be produced.
  ```

- ```
  template<class T>
  void cfGaussSample(
    const ccPelBuffer_const<T>& src, ccPelBuffer<T>& dst,
    const ccGaussSampleParams& params);
  ```

  Computes an image by applying a Gaussian filter to the supplied input image using the supplied **ccGaussSampleParams**.

  This overload is instantiated for 8-bit input and output pel buffers.

  Parameters

  src — The input image. It must be of type **c_UInt8**.

  dst — The image into which the output image is placed. If *dst* is unbound, a root image is allocated and bound. The destination image must be of type **c_UInt8**.

  params — A **ccGaussSampleParams** specifying the sampling and smoothing parameters

  Throws

  ccGaussSampleParams::BadParams
  The sampling rate specified in *params* is such that an output image with a height or width of zero would be produced.

  ccPelFunc::Overlap
  src and *dst* contain one or more pixels in common.

- ```
  template<class T>
  void cfGaussSample(
    const ccPelBuffer_const<T>& src, ccPelBuffer<U>& dst,
    const ccGaussSampleParams& params);
  ```

  Computes an image by applying a Gaussian filter to the supplied input image using the supplied **ccGaussSampleParams**.

  This overload is instantiated for an 8-bit input pel buffer and a 16-bit output pel buffer.

  Parameters

  src — The input image. It must be of type **c_UInt8**.

  dst — The image into which the output image is placed. If *dst* is unbound, a root image is allocated and bound. The destination image must be of type **c_UInt16**.

  params — A **ccGaussSampleParams** specifying the sampling and smoothing parameters

Throws

ccGaussSampleParams::BadParams

The sampling rate specified in *params* is such that an output image with a height or width of zero would be produced.

ccPelFunc::Overlap

src and *dst* contain one or more pixels in common.

cfGaussSample()

cfGenerateOcrChecksum()

```
#include <ch_cvl/acuread.h>

cfGenerateOcrChecksum();
```

Global function to generate a checksum for a string.

cfGenerateOCRChecksum

```
ccCvlString cfGenerateOcrChecksum(
   ccAcuReadDefs::Checksum checksum,
   const ccCvlString &str);
```

Returns the string that you supply (*str*) with a valid checksum appended.

Parameters

checksum The checksum to compute. *checksum* must be one of the following values:

ccAcuReadDefs::eSemi
ccAcuReadDefs::eBC412
ccAcuReadDefs::eIBM412

str The string to compute a checksum for.

Throws

ccAcuReadDefs::BadParams
str contains invalid characters; *checksum* is *ccAcuReadDefs::eSemi* and *str* is longer than *ccAcuReadDefs::kMaxStringLength* -2; *checksum* is *ccAcuReadDefs::eBC412* or *ccAcuReadDefs::eBC412* and *str* is longer than *ccAcuReadDefs::kMaxStringLength* -1; or *checksum* is *ccAcuReadDefs::eNone* or *ccAcuReadDefs::eVirtual*.

cfGenerateOcrChecksum()

cfGetColorRangeFromImage()

```
#include <ch_cvl/coltool.h>
```

`cfGetColorRangeFromImage()`

Calculates the mean color and gets the color range and standard deviations of a three-plane image. See **cfGetColorRangeFromImageRegion()** for a version that uses shapes instead of a mask to define a region.

cfGetColorRangeFromImage

```
void cfGetColorRangeFromImage(
                    const cc3PlanePelBuffer& image,
                    ccColorRange& color,
                    cc3Vect& standardDeviations,
                    const ccColorStatisticsParams &params,
                    double standardDeviationScaleFactor);

void cfGetColorRangeFromImage(
                    const cc3PlanePelBuffer& image,
                    const ccPelBuffer_const<c_UInt8>& mask,
                    ccColorRange& color,
                    cc3Vect& standardDeviations,
                    const ccColorStatisticsParams &params,
                    double standardDeviationScaleFactor);
```

- ```
  void cfGetColorRangeFromImage(
                      const cc3PlanePelBuffer& image,
                      ccColorRange& color,
                      cc3Vect& standardDeviations,
                      const ccColorStatisticsParams &params,
                      double standardDeviationScaleFactor);
  ```

  Calculates the mean color and returns the color range of *image* in *color* and the standard deviations in *standardDeviations*.

  Parameters

  image The three-plane color image.

  color The returned color range.

  standardDeviations
  The returned standard deviations.

  params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation. If omitted, the default values of zero are used.
  ```

standardDeviationScaleFactor
A factor to scale the high and low tolerance of the returned color range. If omitted, the value is 1.0.

Throws

ccColorToolDefs::BadImage
image is not bound

ccColorToolDefs::NotEnoughSamples
Not enough pels (< 1) in the image satisfy the constraints of *params*.

ccColorToolDefs::BadColorSpace
image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.

- ```
  void cfGetColorRangeFromImage(
                      const cc3PlanePelBuffer& image,
                      const ccPelBuffer_const<c_UInt8>& mask,
                      ccColorRange& color,
                      cc3Vect& standardDeviations,
                      const ccColorStatisticsParams &params,
                      double standardDeviationScaleFactor);
  ```

Calculates the mean color and returns the color range of the portion of *image* described by *mask* in *color* and the standard deviations in *standardDeviations*.

Parameters

image The three-plane color image.

mask An image mask. Pels whose value is 0 are don't care pixels; pels whose values are 255 are care pixels. All other values throw an error.

color The returned color range.

standardDeviations
The returned standard deviations.

params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation. If omitted, the default values of zero are used.

standardDeviationScaleFactor
A factor to scale the high and low tolerance of the returned color range. If omitted, the value is 1.0.
  ```

Throws

ccColorToolDefs::BadImage
image or *mask* is not bound
image window is not the same as *mask* window
mask contains pel whose values are other than 0 and 255

ccColorToolDefs::NotEnoughSamples
Not enough pels (< 1) in the image satisfy the constraints of *mask* or *params*.

ccColorToolDefs::BadColorSpace
image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.

cfGetColorRangeFromImage()

cfGetColorRangeFromImageRegion()

```
#include <ch_cvl/coltool.h>
```

`cfGetColorRangeFromImageRegion()`

Gets the color range and standard deviations of a three-plane image. See **cfGetColorRangeFromImage()** for a version that uses masks instead of shapes to define a region.

cfGetColorRangeFromImageRegion

```
void cfGetColorRangeFromImageRegion(
                    const cc3PlanePelBuffer& image,
                    const ccShapePtrh& region,
                    ccColorRange& color,
                    cc3Vect& standardDeviations,
                    const ccColorStatisticsParams &params,
                    double standardDeviationScaleFactor);
```

Calculates the mean color and returns the color range of the intersection of *image* and *region* in *color* and the standard deviations in *standardDeviations*.

Region is defined in image's client coordinate system and can partially overlap *image*. In that case the intersection is used. If more than 50% of a pel falls inside the region, it is considered as being completely inside the region

Parameters

image The three-plane color image.

region A shape that defines a region. If *region* is NULL, the whole image is used.

color The returned color range.

standardDeviations
The returned standard deviations.

params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation. If omitted, the default values of zero are used.

standardDeviationScaleFactor
A factor to scale the high and low tolerance of the returned color range. If omitted, the value is 1.0.

Throws

ccColorToolDefs::BadImage
image is not bound

ccColorToolDefs::NotEnoughSamples
region and *image* do not intersect

Not enough pels (< 1) in the image satisfy the constraints of *params* or *region*.

If only one pel satisfies the constraints of *params* or *region*, the standard deviations in the result are defined to be cc3Vect(0,0,0), and no error is thrown.

ccColorToolDefs::ShapeIsNotRegion
region is not a region shape.

ccColorToolDefs::BadColorSpace
image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.

ccColorToolDefs::NonLinearXform
the image from client transform is nonlinear

cfGetColorStatisticsFromImage()

```
#include <ch_cvl/coltool.h>
```

`cfGetColorStatisticsFromImage()`

Global function to get statistics from a three-plane image. See **cfGetColorRangeFromImageRegion()** for a version that uses shapes instead of a mask to define a region.

cfGetColorStatisticsFromImage

```
void cfGetColorStatisticsFromImage(
            const cc3PlanePelBuffer& image,
            const ccColorStatisticsParams &params,
            ccColorStatisticsResult &result);

void cfGetColorStatisticsFromImage(
            const cc3PlanePelBuffer& image,
            const ccPelBuffer_const<c_UInt8>& mask,
            const ccColorStatisticsParams &params,
            ccColorStatisticsResult &result);
```

- ```
  void cfGetColorStatisticsFromImage(
              const cc3PlanePelBuffer& image,
              const ccColorStatisticsParams &params,
              ccColorStatisticsResult &result);
  ```

  Gets the mean color and standard deviations for *image* and returns it in *result*.

  Parameters

  image The three-plane color image.

  params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation.

  result a **ccColorStatisticsResult** that contains the computed mean color and standard deviations.

  Throws

  ccColorToolDefs::BadImage
  image is not bound

  ccColorToolDefs::NotEnoughSamples
  Not enough pels (< 1) in the image satisfy the constraints of *params*.

  If only one pel satisfies the constraints of params, the standard deviations in the result are defined to be cc3Vect(0,0,0), and no error is thrown.
  ```

ccColorToolDefs::BadColorSpace
image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.

- ```
  void cfGetColorStatisticsFromImage(
              const cc3PlanePelBuffer& image,
              const ccPelBuffer_const<c_UInt8>& mask,
              const ccColorStatisticsParams &params,
              ccColorStatisticsResult &result);
  ```

Gets the mean color and standard deviations for *image* and returns it in *result*.

Parameters

image The three-plane color image.

mask An image mask. Pels whose value is 0 are don't care pixels; pels whose values are 255 are care pixels. All other values throw an error.

params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation.

result a **ccColorStatisticsResult** that contains the computed mean color and standard deviations.

Throws

ccColorToolDefs::BadImage
image or *mask* is not bound
image window is not the same as *mask* window
mask contains pel whose values are other than 0 and 255

ccColorToolDefs::NotEnoughSamples
Not enough pels (< 1) in the image satisfy the constraints of *mask* or *params*.

If only one pel satisfies the constraints of mask or params, the standard deviations in the result are defined to be cc3Vect(0,0,0), and no error is thrown.

ccColorToolDefs::BadColorSpace
image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.
  ```

cfGetColorStatisticsFromImageRegion()

```
#include <ch_cvl/coltool.h>
```

`cfGetColorStatisticsFromImageRegion()`

Global function to get statistics from a three-plane image. See **cfGetColorStatisticsFromImage()** for a version that uses masks instead of shapes to define a region.

cfGetColorStatisticsFromImageRegion

```
void cfGetColorStatisticsFromImageRegion(
          const cc3PlanePelBuffer& image,
          const ccShapePtrh& region,
          const ccColorStatisticsParams &params,
          ccColorStatisticsResult &result);
```

Gets the mean color and standard deviations for the intersection of *image* and *region*, and returns it in *result*.

Region is defined in image's client coordinate system and can partially overlap *image*. In that case the intersection is used. If more than 50% of a pel falls inside the region, it is considered as being completely inside the region

Parameters

image The three-plane color image.

region A shape that defines a region. If *region* is NULL, the whole image is used.

params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation.

result a **ccColorStatisticsResult** that contains the computed mean color and standard deviations.

Throws

ccColorToolDefs::BadImage
image is not bound

ccColorToolDefs::NotEnoughSamples
region and *image* do not intersect

Not enough pels (< 1) in the image satisfy the constraints of *params* or *region*.

If only one pel satisfies the constraints of *params* or *region*, the standard deviations in the result are defined to be cc3Vect(0,0,0), and no error is thrown.

ccColorToolDefs::ShapeIsNotRegion
: *region* is not a region shape.

ccColorToolDefs::BadColorSpace
: *image's* color space is **ccColorSpaceDefs::eUnknown** or illegal value.

ccColorToolDefs::NonLinearXform
: the image from client transform is nonlinear

cfGetCompileTimeCvlVersion()

```
#include <ch_cvl/version.h>

cfGetCompileTimeCvlVersion();
```

Global function that retrieves the CVL version used at compile time.

cfGetCompileTimeCvlVersion

```
ccVersion cfGetCompileTimeCvlVersion();
```

Retrieves the CVL version used for compilation.

cfGetCompileTimeCvlVersion()

cfGetCurrentThreadID()

```
#include <ch_cvl/threads.h>

cfGetCurrentThreadID();
```

Global function to return a **ccThreadID** that refers to the current thread.

cfGetCurrentThreadID

```
ccThreadID cfGetCurrentThreadID();
```

Notes

You can use the returned **ccThreadID** to set a thread termination callback function, to break all the locks on a thread (by calling the static function **cc_Resource::breakLocks()**), or to block one thread until another thread completes.

cfGetCurrentThreadID()

cfGetRunTimeCvlVersion()

```
#include <ch_cvl/version.h>
```

```
cfGetRunTimeCvlVersion();
```

Global function that retrieves the CVL version used at run time.

cfGetRunTimeCvlVersion

```
ccVersion cfGetRunTimeCvlVersion();
```

Returns the CVL version used at run time. If the vision application is linked against CVL static libraries, then this function returns the compile time version.

cfGetRunTimeCvlVersion()

cfGetThreadPriority()

```
#include <ch_cvl/threads.h>

cfGetThreadPriority();
```

Global function to return the thread priority of the current thread.

cfGetThreadPriority

```
cePriority cfGetThreadPriority ();
```

cfGetThreadPriority() returns one of the following values:

cePriorityDefault
cePriorityIdle
cePriorityLowest
cePriorityBelowNormal
cePriorityNormal
cePriorityAboveNormal
cePriorityHighest
cePriorityTimeCritical

cfGetThreadPriority()

cfGetSimpleColorFromImage()

```
#include <ch_cvl/coltool.h>
```

`cfGetSimpleColorFromImage()`

Gets the mean color from a color image.

cfGetSimpleColorFromImage

```
void cfGetSimpleColorFromImage(
                    const cc3PlanePelBuffer& image,
                    ccColorValue& meanColor,
                    const ccColorStatisticsParams &params);

void cfGetSimpleColorFromImage(
                    const cc3PlanePelBuffer& image,
                    const ccPelBuffer_const<c_UInt8>& mask,
                    ccColorValue& meanColor,
                    const ccColorStatisticsParams &params);
```

- ```
  void cfGetSimpleColorFromImage(
                      const cc3PlanePelBuffer& image,
                      ccColorValue& meanColor,
                      const ccColorStatisticsParams &params);
  ```

  Returns, in *meanColor*, the mean color of *image*.

  Parameters

  image The color image

  meanColor The returned mean color in *image*. The mean color has the same color space as *image*.

  params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation. If omitted, the default values of zero are used.

  Throws

  ccColorToolDefs::BadImage
  image is not bound

  ccColorToolDefs::NotEnoughSamples
  Not enough pels (< 1) in the image satisfy the constraints of *params*.

  ccColorToolDefs::BadColorSpace
  image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.
  ```

cfGetSimpleColorFromImage()

- ```
  void cfGetSimpleColorFromImage(
                      const cc3PlanePelBuffer& image,
                      const ccPelBuffer_const<c_UInt8>& mask,
                      ccColorValue& meanColor,
                      const ccColorStatisticsParams &params);
  ```

  Returns, in *meanColor*, the mean color of *image* described by *mask*.

  Parameters

  image — The color image

  mask — An image mask. Pels whose value is 0 are don't care pixels; pels whose values are 255 are care pixels. All other values throw an error.

  meanColor — The returned mean color in *image*. The mean color has the same color space as *image*.

  params — a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation. If omitted, the default values of zero are used.

  Throws

  ccColorToolDefs::BadImage
  image or *mask* is not bound
  image window is not the same as *mask* window
  mask contains pel whose values are other than 0 and 255

  ccColorToolDefs::NotEnoughSamples
  Not enough pels (< 1) in the image satisfy the constraints of *mask* or *params*.

  ccColorToolDefs::BadColorSpace
  image's color space is **ccColorSpaceDefs::eUnknown** or illegal value.
  ```

cfGetSimpleColorFromImageRegion()

```
#include <ch_cvl/coltool.h>
```

`cfGetSimpleColorFromImageRegion()`

Gets the mean color from an image. See **cfGetSimpleColorFromImage()** for a version that uses masks instead of shapes to define a region

cfGetSimpleColorFromImageRegion

```
void cfGetSimpleColorFromImageRegion(
                    const cc3PlanePelBuffer& image,
                      const ccShapePtrh& region,
                      ccColorValue& meanColor,
                      const ccColorStatisticsParams &params);
```

Gets the mean color for the intersection of *image* and *region*, and returns it in *meanColor*

Region is defined in image's client coordinate system and can partially overlap *image*. In that case the intersection is used. If more than 50% of a pel falls inside the region, it is considered as being completely inside the region

Parameters

image The three-plane color image.

region A shape that defines a region. If *region* is NULL, the whole image is used.

meanColor The returned mean color in *image*. The mean color has the same color space as *image*.

params a **ccColorStatisticsParams** object to specify the minimum intensity and minimum saturation. If omitted, the default values of zero are used.

Throws

ccColorToolDefs::BadImage
image is not bound

ccColorToolDefs::NotEnoughSamples
region and *image* do not intersect

Not enough pels (< 1) in the image satisfy the constraints of *params* or *region*.

If only one pel satisfies the constraints of *params* or *region*, the standard deviations in the result are defined to be cc3Vect(0,0,0), and no error is thrown.

ccColorToolDefs::ShapeIsNotRegion
: *region* is not a region shape.

ccColorToolDefs::BadColorSpace
: *image's* color space is **ccColorSpaceDefs::eUnknown** or illegal value.

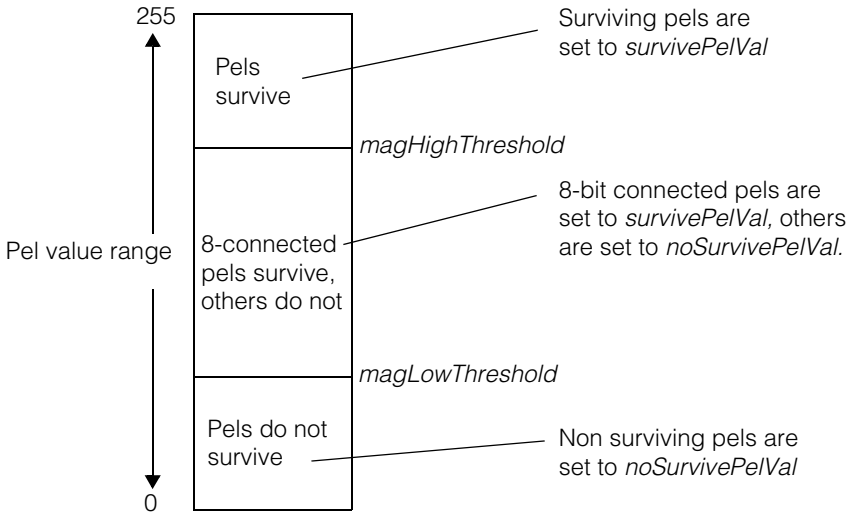
ccColorToolDefs::NonLinearXform
: the image from client transform is nonlinear

cfHysteresisThreshold()

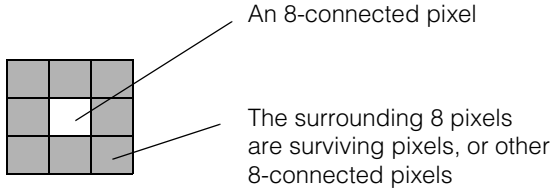
```
#include <ch_cvl/edge.h>

cfHysteresisThreshold();
```

This function performs hysteresis thresholding on the supplied edge magnitude image. The result image contains a binarized edge map. The following diagram shows the primary parameters used by the function to produce pixels in the binarized result. This example is for 8-bit pel values.



All pixels in the supplied image with values less than *magLowThreshold* do not survive and are set to *noSurvivePelVal*. All pixels in the supplied image with values greater than or equal to *magHighThreshold* are set to *survivePelVal*. All pixels in the supplied image with values greater than or equal to *magLowThreshold* and less than *magHighThreshold* are set to *survivePelVal* if they are 8-connected to a pixel with a value greater than or equal to *magHighThreshold*, either directly or through other 8-connected pixels. The following diagram shows an 8-bit connected pixel.



Pixels with values greater than or equal to *magLowThreshold* but less than *magHighThreshold* that are not 8-connected do not survive and are set to *noSurvivePelVal*.

cfHysteresisThreshold()

cfHysteresisThreshold

```
template<class T>
void cfHysteresisThreshold(
  const ccPelBuffer_const<T>& magImage,
  ccPelBuffer<c_UInt8>& dst,
  T magLowThreshold,
  T magHighThreshold,
  c_Int32* nSurvivors = 0,
  c_Int32 nPels = -1,
  T noSurvivePelVal = 0,
  T survivePelVal = 1);

template<class T>
  ccPelBuffer<c_UInt8> cfHysteresisThreshold(
  const ccPelBuffer_const<T>& magImage,
  T magLowThreshold,
  T magHighThreshold,
  c_Int32* nSurvivors = 0,
  c_Int32 nPels = -1,
  T noSurvivePelVal = 0,
  T survivePelVal = 1);
```

-
```
template<class T>
void cfHysteresisThreshold(
  const ccPelBuffer_const<T>& magImage,
  ccPelBuffer<c_UInt8>& dst,
  T magLowThreshold,
  T magHighThreshold,
  c_Int32* nSurvivors = 0,
  c_Int32 nPels = -1,
  T noSurvivePelVal = 0,
  T survivePelVal = 1);
```

Performs hysteresis thresholding on the supplied input edge magnitude image (*magImage*) and places the result in the (*dst*) image you supply.

Notes

Although this is a templated function, the current implementation only handles **c_UInt8**.

Parameters

magImage	The input edge magnitude image. *magImage* must be of type **c_UInt8**.
dst	The image produced by edge hysteresis thresholding. If *dst* is unbound, a root image is allocated and bound. *dst* must be of type **c_UInt8**.

magLowThreshold

The low threshold

magHighThreshold

The high threshold

nSurvivors If *nSurvivors* is not *NULL*, the number of nonzero pixels in the returned image is written to the **c_Int32** pointed to by *nSurvivors*.

nPels If the number of nonzero pixels in *magImage* is known, you can reduce the amount of memory required for this function by passing this value in *nPels*.

If you do not know the number of nonzero pixels in *magImage*, set *nPels* equal to -1.

noSurvivePelVal The pixel value assigned to non-surviving pels.

survivePelVal The pixel value assigned to surviving pels.

- ```
  template<class T>
   ccPelBuffer<c_UInt8> cfHysteresisThreshold(
   const ccPelBuffer_const<T>& magImage,
   T magLowThreshold,
   T magHighThreshold,
   c_Int32* nSurvivors = 0,
   c_Int32 nPels = -1,
   T noSurvivePelVal = 0,
   T survivePelVal = 1);
  ```

  Performs hysteresis thresholding on the supplied input edge magnitude image (*magImage*) and returns the resulting image.

  Notes

  Although this is a templated function, the current implementation only handles **c_UInt8**.

  Parameters

  magImage The input edge magnitude image. *magImage* must be of type **c_UInt8**.

  magLowThreshold

  The low threshold

  magHighThreshold

  The high threshold
  ```

nSurvivors	If *nSurvivors* is not *NULL*, the number of nonzero pixels in the returned image is written to the **c_Int32** pointed to by *nSurvivors*.
nPels	If the number of nonzero pixels in *magImage* is known, you can reduce the amount of memory required for this function by passing this value in *nPels*. If you do not know the number of nonzero pixels in *magImage*, set *nPels* equal to -1.
noSurvivePelVal	The pixel value assigned to non-surviving pels.
survivePelVal	The pixel value assigned to surviving pels.

cfIDDecode()

```
#include <ch_cvl/id.h>

cfIDDecode();
```

Global function to find, decode, and grade symbols in an image.

cfIDDecode

```
void cfIDDecode(const ccPelBuffer_const<c_UInt8>& image,
      const ccIDSearchParams& searchParams,
      const ccIDDecodeParams& decodeParams,
      ccIDResultSet& resultSet, ccDiagObject* diagobj = 0,
      c_UInt32 diagFlags = 0);
```

Searches for and decodes symbols in the supplied image using the supplied search and decode parameters.

Parameters

image	The image to search.
searchParams	The search parameters.
decodeParams	The decoding parameters.
resultSet	The result set into which the results are placed.
diagobj	A container class object that holds diagnostic records created during the run of the tool. If you supply a value, the tool records diagnostic information as specified by *diagFlags*.
diagFlags	Set to 0 to record no diagnostic information. Otherwise *diagFlags* is composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

Throws

ccIDDefs::BadParams
: No symbology is enabled in *decodeParams*.

ccIDDefs::BadImage
: *image* is unbound.

ccIDDefs::ImageTooSmall
: *image* is smaller than 16x16 pixels.

ccTimeout::Expired
: The user-specified timeout expires. Any completed results are returned in *resultSet*.

ccIDDefs::InternalError
: An unexpected error occurred.

cfImageRegister()

```
#include <ch_cvl/imgreg.h>

cfImageRegister();
```

Global function invoke the Image Registration tool. The tool will locate the reference image within the source image, given an appropriate starting location.

cfImageRegister

```
void cfImageRegister(const ccPelBuffer_const<T>& src,
  const ccPelBuffer_const<T>& ref,
  const ccImageRegisterParams& params,
  ccImageRegisterResults &results);

void cfImageRegister(const ccPelBuffer_const<T>& src,
  const ccPelBuffer_const<T>& ref,
  const cmStd vector<ccPelRect> &rects,
  const ccImageRegisterParams& params,
  ccImageRegisterResults &results);

ccImageRegisterResults cfImageRegister(
  const ccPelBuffer_const<T>& src,
  const ccPelBuffer_const<T>& ref,
  const ccImageRegisterParams& params);

ccImageRegisterResults cfImageRegister(
  const ccPelBuffer_const<T>& src,
  const ccPelBuffer_const<T>& ref,
  const cmStd vector<ccPelRect> &rects,
  const ccImageRegisterParams& params);
```

- ```
  void cfImageRegister(const ccPelBuffer_const<T>& src,
    const ccPelBuffer_const<T>& ref,
    const ccImageRegisterParams& params,
    ccImageRegisterResults &results);
  ```

  Locates the supplied reference image within the supplied source image using the supplied parameters.

  Parameters

src	The source image. *src* can be of type **c_UInt8**.
ref	The reference image. *ref* must be smaller than, and of the same type as, *src*.
params	A **ccImageRegisterResults** containing the parameters to use for this image registration.
results	A **ccImageRegisterResults** into which the results are placed.
  ```

Throws

ccImgRegDefs::UnboundWindow
: *src* or *ref* is unbound.

ccImgRegDefs::BadStartPosition
: When placed at the starting position specified in *params*, the reference image is not completely contained within *src*.

ccImgRegDefs::ModelNotContainedInSource
: *ref* is either taller or wider than *src*.

- ```
  void cfImageRegister(const ccPelBuffer_const<T>& src,
    const ccPelBuffer_const<T>& ref,
    const cmStd vector<ccPelRect> &rects,
    const ccImageRegisterParams& params,
    ccImageRegisterResults &results);
  ```

Locates the supplied reference image within the supplied source image using the supplied parameters.

Only reference image pixels which lie within the supplied rectangles are considered.

Parameters

src	The source image. *src* can be of type **c_UInt8**.
ref	The reference image. *ref* must be smaller than, and of the same type as, *src*.
rects	A vector of rectangles, specified in the reference image's image coordinate system. Only pixels within these rectangles are considered by the tool. If a pixel is contained within more than one rectangle, it is evaluated once for each rectangle that contains it.
params	A **ccImageRegisterResults** containing the parameters to use for this image registration.
results	A **ccImageRegisterResults** into which the results are placed.

Throws

ccImgRegDefs::UnboundWindow
: *src* or *ref* is unbound.

ccImgRegDefs::BadStartPosition
: When placed at the starting position specified in *params*, the reference image is not completely contained within *src*.
  ```

ccImgRegDefs::ModelNotContainedInSource
ref is either taller or wider than *src*.

ccImgRegDefs::RectNotContainedInModel
One or more of the rectangles in *rects* is not completely contained within *ref*.

- ```
  ccImageRegisterResults cfImageRegister(
    const ccPelBuffer_const<T>& src,
    const ccPelBuffer_const<T>& ref,
    const ccImageRegisterParams& params);
  ```

  Locates the supplied reference image within the supplied source image using the supplied parameters.

  Parameters

  src The source image. *src* can be of type **c_UInt8**.

  ref The reference image. *ref* must be smaller than, and of the same type as, *src*.

  params A **ccImageRegisterResults** containing the parameters to use for this image registration.

  results A **ccImageRegisterResults** into which the results are placed.

  Throws

  ccImgRegDefs::UnboundWindow
  src or *ref* is unbound.

  ccImgRegDefs::BadStartPosition
  When placed at the starting position specified in *params*, the reference image is not completely contained within *src*.

  ccImgRegDefs::ModelNotContainedInSource
  ref is either taller or wider than *src*.

- ```
  ccImageRegisterResults cfImageRegister(
    const ccPelBuffer_const<T>& src,
    const ccPelBuffer_const<T>& ref,
    const cmStd vector<ccPelRect> &rects,
    const ccImageRegisterParams& params);
  ```

  Locates the supplied reference image within the supplied source image using the supplied parameters.

  Only reference image pixels which lie within the supplied rectangles are considered.

cfImageRegister()

Parameters

src The source image. *src* can be of type **c_UInt8**.

ref The reference image. *ref* must be smaller than, and of the same type as, *src*.

rects A vector of rectangles, specified in the reference image's image coordinate system.

Only pixels within these rectangles are considered by the tool. If a pixel is contained within more than one rectangle, it is evaluated once for each rectangle that contains it.

params A **ccImageRegisterResults** containing the parameters to use for this image registration.

Throws

ccImgRegDefs::UnboundWindow
src or *ref* is unbound.

ccImgRegDefs::BadStartPosition
When placed at the starting position specified in *params*, the reference image is not completely contained within *src*.

ccImgRegDefs::ModelNotContainedInSource
ref is either taller or wider than *src*.

ccImgRegDefs::RectNotContainedInModel
One or more of the rectangles in *rects* is not completely contained within *ref*.

cfImageSharpness()

```
#include <ch_cvl/imgsharp.h>

cfImageSharpness();
```

Global function to determine the sharpness of an image. You call this function to run the Image Sharpness tool.

The primary purpose of this tool is to automate focusing a camera lens on a scene. You acquire an image and run the tool to obtain a sharpness score. You then refocus the lens changing nothing else, acquire another image and score it. By iteratively refocusing, scoring, and comparing the sharpness scores you can home in on the highest score which is the best camera focus you can obtain for your application. A second global function, **cfImageSharpnessFocusSearch()**, is provided to help you implement this iterative procedure.

Note that a single sharpness score conveys no information about the absolute sharpness of an image. Also, comparing sharpness scores of different scenes has no meaning.

When you call **cfImageSharpness()** you pass it an image to score, and a parameters object that specifies how the score should be calculated. The scoring parameters specify one of five algorithms which are discussed below:

	Score range		
Algorithm	**Low (Blurred image)**	**High (Sharp image)**	**Parameters used**
eAutoCorrelation	0	1000	*noiseLevel*
eAbsDiff	0	255	*sx, sy*
eEdgeGradient **Note**: This algorithm is deprecated	0	255	
eBandPass	0	No upper limit	***freqBand***
eGradientEnergy[1]	0	No upper limit	***lowPassSmoothing***

(1) Most applications achieve the best results using this algorithm.

Notes

1. *eAutoCorrelation* algorithm:

   The sharpness score is based on the local variation of the autocorrelation function. When correlation scores are high (near 1.0) the image is blurry. When correlation scores are low (near 0) the image has sharp edges. The following formula is used to calculate the auto-correlation score where r is the composite correlation score for the image.

   $$1000 - (1000r)$$

2. *eAbsDiff* algorithm:

   The sharpness score is based on the selective sum of absolute differences of image samples separated by specified distances *sx* and *sy* in x and y directions respectively. You set *sx* and *sy* in the *params* object based on your knowledge of the underlying pattern. (For example, the frequency of interest you wish to resolve).

   Pixel intensity is compared at (x,y) and (x, y+sy), then (x,y) and (x+sx, y), for all pixels in the image valid region. The image sharpness score is the mean of the average sharpness in the x-direction and the average sharpness in the y-direction over the entire valid region.

3. *eEdgeGradient* algorithm: (**Note**: This algorithm is deprecated)

   The sharpness score is based on the average edge gradient in the image. In *eEdgeGradient* mode **cfImageSharpness()** applies a Sobel edge operator to the source image and computes magnitudes of all edge gradients in the image. The mean of these edge magnitudes is returned as the sharpness score.

4. *eBandPass* algorithm:

   The sharpness score is based on frequencies occurring in the band of frequencies, **ccSharpnessParams::freqBand()**. Sharp edges produce high frequencies and blurred edges produce low frequencies. You choose this frequency band based on expected frequencies occurring in the image.

5. *eGradientEnergy* algorithm:

   The sharpness score is based on the sum of the squares of the edge gradients. See the *Image Sharpness Tool* chapter of the *CVL Vision Tools Guide* for information about how this algorithm works.

6. The same image may yield very different sharpness scores depending on which algorithm is used. Also, comparison of sharpness scores of multiple images using the same algorithm is meaningful only if the images used are of the same physical scene and differ only in sharpness or focus settings. For example, if you use the *eAutoCorrelation* algorithm to obtain sharpness scores for images of two different scenes, there is no relationship between the two scores. The image with the highest score in not necessarily sharper than the other image.
7. The **cfImageSharpness()** function does not use the client transform in the image.

For a detailed discussion of the sharpness algorithms see the *Image Sharpness Tool* chapter of the *Vision Tools Guide*.

cfImageSharpness

```
double cfImageSharpness(
  const ccPelBuffer_const<c_UInt8>& src,
  const ccSharpnessParams& params);

double cfImageSharpness(
  const ccPelBuffer_const<c_UInt8>& src,
  const ccPelBuffer_const<c_UInt8>& mask,
  const ccSharpnessParams& params);

double cfImageSharpness(
  const ccPelBuffer_const<c_UInt16>& src,
  const ccSharpnessParams& params);
```

Returns a sharpness score for input image (*src*) using the sharpness parameters (*params*). See the introduction to this reference page and the *Image Sharpness Tool* chapter of the *Vision Tools Guide* for information on how this tool works.

Notes

The client coordinate transforms of *src* and *mask* are not used.

Errors (**Throws**) are dependent on the overload called, and the algorithm specified in *params*. The following **Throws** are shown by algorithm. Unless noted, these throws apply to all three **cfImageSharpness()** overloads.

1. eAutoCorrelation

Throws

ccPel::BadWindow
If the *src* image is smaller than 3x3 pels.

ccImageSharpnessDefs::NotImplemented
If the overload with mask is called.

2. eAbsDiff

Throws

ccPel::BadWindow
If either *src* image dimension is 0.

ccPel::BadCoord(sx,sy)
If *sx* and *sy* do not lie within the valid range. Valid ranges are:
sx range (0 through **src.width()** - 1)
sy range (0 through **src.height()** - 1)

ccImageSharpnessDefs::NotImplemented
If the overload with mask is called.

3. *eEdgeGradient* (Deprecated)

Throws

ccPel::BadWindow
If the *src* image is smaller than 3x3 pels.

ccImageSharpnessDefs::NotImplemented
If the overload with mask is called.

4. eBandPass

Throws

ccImageSharpnessDefs::InternalError
If the sharpness score could not be computed.

ccImageSharpnessDefs::UnboundWindow
If the image is unbound.

ccImageSharpnessDefs::NotImplemented
If the overload with mask is called.

5. eGradientEnergy

Throws

ccImageSharpnessDefs::ImageTooSmall
If the sharpness score could not be computed because of small image dimensions.

ccImageSharpnessDefs::UnboundWindow
If either the *src* image or *mask* is unbound.

ccImageSharpnessDefs::BadMask
If the *mask* image dimensions are different than the *src* image dimensions.

ccImageSharpnessDefs::NotImplemented
If the overload with the `c_UInt16` image is called.

- ```
  double cfImageSharpness(
    const ccPelBuffer_const<c_UInt8>& src,
    const ccSharpnessParams& params);
  ```

  Parameters

  src The image to score.

  params The sharpness parameters.

- ```
  double cfImageSharpness(
    const ccPelBuffer_const<c_UInt8>& src,
    const ccPelBuffer_const<c_UInt8>& mask,
    const ccSharpnessParams& params);
  ```

  Notes

  This overload is only supported for the *eGradientEnergy* algorithm.

  The *mask* image must have the same dimensions as the *src* image. *src* image pixels are used if the corresponding *mask* pixel value is 255. *src* image pixels are ignored if the corresponding *mask* pixel value is 0. All other *mask* pixel values are invalid.

  Parameters

  src The image to score.

  mask Apply this mask to the *src* image.

  params The sharpness parameters.

- ```
  double cfImageSharpness(
    const ccPelBuffer_const<c_UInt16>& src,
    const ccSharpnessParams& params);
  ```

  Notes

  This overload is not supported for the *eGradientEnergy* algorithm.

  Parameters

  src The image to score.

  params The sharpness parameters.
  ```

Deprecated Members

The following functions are deprecated and included here for backward compatibility only.

cfImageSharpness

```
template<class T>
T cfImageSharpness(const ccPelBuffer_const<T>& src,
  c_UInt32 sx, c_UInt32 sy);

template<class T>
  c_Int16 cfImageSharpness(
  const ccPelBuffer_const<T>& src);
```

```
template<class T>
T cfImageSharpness(const ccPelBuffer_const<T>& src,
  c_UInt32 sx, c_UInt32 sy);
```

Determines the image sharpness using image difference mode at the specified x and y offset. Image sharpness is measured by computing the average difference in pixel intensity value between each pixel in the input image and the pixel within the image offset by the specified amount.

The higher the returned value, the sharper the image. Return values can be in the range 0 to the maximum value supported by the template type *T*.

Parameters

src	The input image.
sx	The x offset, in image coordinate units, at which to measure image difference.
sy	The y offset, in image coordinate units, at which to measure image difference.

-
```
template<class T>
  c_Int16 cfImageSharpness(
  const ccPelBuffer_const<T>& src);
```

Determines the image sharpness using auto-correlation mode. Image blurriness is measured by computing the correlation coefficient at 1-pixel offsets. The highest correlation coefficient is the image blurriness. This function returns a value equal to

$$1000 - (1000r)$$

where r is the image blurriness. The returned value can be in the range 0 to 1000.

Parameters

src	The input image.

cfImageSharpness()

cfImageSharpnessFocusSearch()

```
#include <ch_cvl/imgsharp.h>

cfImageSharpnessFocusSearch();
```

This is a global function you can use along with **cfImageSharpness()** to find the number of motor steps required to reach the best camera position (sharpest image focus) within a specified tolerance. This function searches for and finds the optimal depth of focus. You may wish to call this function in your application initialization code to focus your camera before beginning a production run.

To use the function you must write a routine that controls your application and will physically move the camera focus to a new location. You also must write a routine that acquires an image at the new focus and evaluates the image sharpness at this location. It is intended that you call **cfImageSharpness()** to obtain the image sharpness score.

The function is defined by two overloads that both accomplish the same objective. The first overload uses a focus range from 0 through a maximum, measured in focus increments. The second overload defines the focus range with minimum and maximum values. Also, the first overload requires that you write the move and sharpness evaluation routines separately, whereas the second overload combines these into one function that also takes a parameter you can provide.

cfImageSharpnessFocusSearch

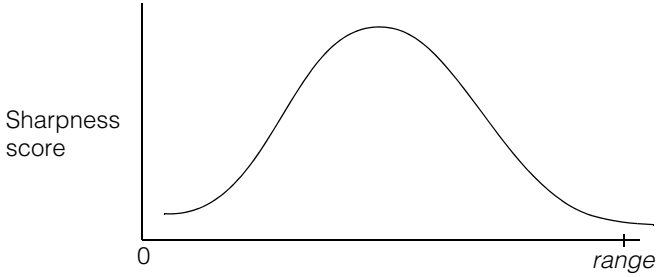
```
c_Int32 cfImageSharpnessFocusSearch (
  c_Int32 range,
  c_Int32 tolerance,
  c_Int32 (*evaluateSharpness)(),
  void (*move)(c_Int32 position));

c_Int32 cfImageSharpnessFocusSearch(
  c_Int32 rangeMin,
  c_Int32 rangeMax,
  c_Int32 tolerance,
  void* userParam,
  c_Int32 (*moveAndEval)
                    (void* userParam, c_Int32 position));
```

- ```
  c_Int32 cfImageSharpnessFocusSearch (
    c_Int32 range,
    c_Int32 tolerance,
    c_Int32 (*evaluateSharpness)(),
    void (*move)(c_Int32 position));
  ```

  Returns an integer which indicates the number of motor steps required to reach the best camera position (sharpest image focus) for the specified tolerance.

  The search strategy iteratively calls a sharpness evaluation function that you write and a camera move function you also write to hunt down the best depth of focus within the specified tolerance. This strategy works best when the depth of focus range specified results in a unimodal distribution of image sharpness coefficients. For example:

  Working within the range (0 through *range*), **cfImageSharpnessFocusSearch()** iteratively calls the *move* function to move the camera and calls the sharpness function to obtain the sharpness score at various positions. As it homes in on the best score position, the camera movements become smaller and smaller. When the camera movements become smaller than *tolerance*, the procedure ends and the function returns.

  Due to the nature of the algorithm *range* must be less than 701408733.

  Parameters

  range — The number of steps (in depth of focus increments) that the search is performed over. The search starts at zero and will move the camera up to *range* steps to find the best focus within this range. The sharpest focus point must lie within this range for **cfImageSharpnessFocusSearch()** to be successful.

  tolerance — The least number of depth of focus increments by which the position (depth of focus) should change in order to end the search. During the search, if the change in the position does not change by at least *tolerance* increments, the search ends.

  evaluateSharpness — A pointer to a sharpness evaluation function you write. The sharpness evaluation function takes no arguments. It returns an
  ```

image sharpness score (for example, using **cfImageSharpness()**) computed from the image at the current depth of focus. Generally, this is a wrapper function you write that calls **cfImageSharpness()**.

move — A pointer to a function which performs the necessary action to change the camera depth of focus as required during the search. It takes one argument (*position*) which specifies the absolute position where the camera should be set.

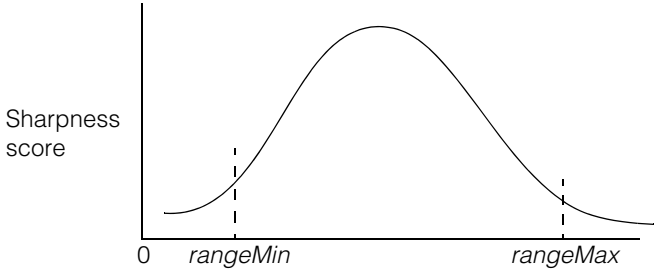
Throws

ccImageSharpnessDefs::BadParams
If *range* < *0*,
or if *range* >= 701408733,
or if *tolerance* < 1,
or if *tolerance* >= *range*.

- ```
  c_Int32 cfImageSharpnessFocusSearch(
    c_Int32 rangeMin,
    c_Int32 rangeMax,
    c_Int32 tolerance,
    void* userParam,
    c_Int32 (*moveAndEval)
                        (void* userParam, c_Int32 position));
  ```

Returns an integer which indicates the number of motor steps required to reach the best camera position (sharpest image focus) for the specified tolerance.

The search strategy iteratively calls a move and evaluation function that you write to hunt down the best depth of focus within the specified tolerance. This strategy works best when the depth of focus range specified results in a unimodal distribution of image sharpness coefficients. For example:



Working within the range (*rangeMin* through *rangeMax*), **cfImageSharpnessFocusSearch()** iteratively calls the *moveAndEval* function to move the camera and obtain sharpness scores at various positions in the range. As it homes
  ```

in on the best score position, the camera movements become smaller and smaller. When the camera movements become smaller than *tolerance*, the procedure ends and the function returns.

Due to the nature of the algorithm *rangeMax-rangeMin* must be less than 701408733.

Parameters

rangeMin — The beginning of the range (in depth of focus increments) where the search is performed. The search starts at *rangeMin* and will move the camera up to *rangeMax* to find the best focus within this range. The sharpest focus point must lie within this range for **cfImageSharpnessFocusSearch()** to be successful.

rangeMax — The end of the range (in depth of focus increments) where the search is performed. The search starts at *rangeMin* and will move the camera up to *rangeMax* to find the best focus within this range. The sharpest focus point must lie within this range for **cfImageSharpnessFocusSearch()** to be successful.

tolerance — The least number of depth of focus increments by which the position (depth of focus) should change in order to end the search. During the search, if the change in the position does not change by at least *tolerance* increments, the search ends.

userParam — A user parameter you define that is passed to the *moveAndEval* function you write. See below.

moveAndEval — A pointer to a function you write that moves the camera focus to a new location, and then evaluates the sharpness of an image acquired at this new location. *position* defines the new location in focus increments measured from 0, and *userParam* is a pointer to a parameter you define that is passed to your routine.

The sharpness evaluation function returns an image sharpness score (for example, by using **cfImageSharpness()**) computed from the image at the current depth of focus. In most cases your *moveAndEval* function should call **cfImageSharpness()** to obtain the sharpness score.

Throws

ccImageSharpnessDefs::BadParams

If *rangeMax* < *rangeMin*,
or if *rangeMax-rangeMin* >= 701408733,
or if *tolerance* < 1,
or if *tolerance* >= *rangeMax-rangeMin*.

cfInitializeDisplayResources()

```
#include <ch_cvl/windisp.h>

cfInitializeDisplayResources();
```

Global function to initialize the resources used by **ccDisplayConsole** and **ccWin32Display** when used from within a DLL.

cfInitializeDisplayResources

```
void cfInitializeDisplayResources();
```

You must call this function to initialize GUI resources before using a **ccDisplayConsole** or **ccWin32Display** from within a DLL.

Notes

If you create your own DLL that initializes CVL display objects and calls their functions, then you must call **cfInitializeDisplayResources()** in your application's initialization and setup code.

Your application *must* call **cfInitializeDisplayResources()** if your application instantiates a **ccWin32Display** or **ccDisplayConsole** object in a DLL instead of in your application's executable.

In all other cases, the work performed by **cfInitializeDisplayResources()** is redundant and benign, but your application can call it without producing an error.

cfInitializeDisplayResources()

cfLabelHistogram()

```
#include <ch_cvl/lablproj.h>

cfLabelHistogram();
```

Global function that computes the labeled histogram of an image.

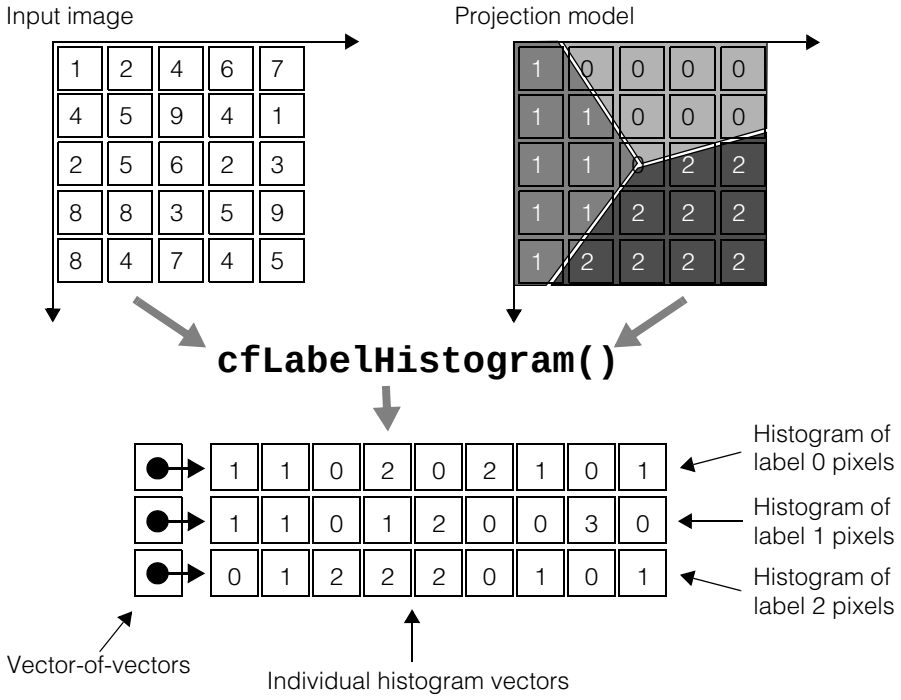
cfLabelHistogram

```
template<class M, class I>
void cfLabelHistogram(
  const ccLabeledProjectionModel<M> &model,
  const ccPelBuffer_const<I> &image,
  cmStd vector<cmStd vector<c_Uint32>> &histograms,
  bool preClearHist = true);
```

Computes the labeled histogram of *image* using the supplied **ccLabeledProjectionModel**.

A labeled histogram is actually a collection of individual histograms, one for each label value defined in the projection model. The function collects all of the pixels which have a given label, then computes the histogram of those pixels. The function places the labeled histogram in the vector of vectors that you supply by reference to the function. The individual vectors within the vector-of-vectors are the individual histograms.

The following figure shows how **cfLabelHistogram** computes a labeled histogram from a simple input image and projection model:



The size of the vector-of-vectors is equal to the largest label value in the projection model. Note that the individual histogram vectors actually have 256 elements, even though only the first 9 elements have nonzero values.

Parameters

M and I Template parameters specifying the type of the projection model and source image. You must specify one of the following combinations of types:

M	I
c_UInt8	**c_UInt8**
c_UInt16	**c_UInt8**

model The labeled projection model to use.

image The input image.

histograms A vector of histograms. If **histograms.size()** is zero, it is resized to **model.projectionLength()**.

Any individual histograms within *histograms* that have a length of zero are resized to have a length of 256 (for 8-bit images) or to have a length equal to the largest pixel value for the input image.

Individual histograms that have a length greater than 256 are resized to have a length of 256.

preClearHist If *preClearHist* is true, this function will initialize the contents of all the individual histograms to zero. If *preClearHist* is false, then the existing histogram contents are added to.

Throws

ccLabeledProjection::HistogramTooShort
histograms.size() is greater than zero but less than **model.projectionLength()**.

cfLabelHistogram()

cfLabelProject()

```
#include <ch_cvl/lablproj.h>

cfLabelProject();
```

Global function to compute both a normalized and non-normalized projection image using the supplied **ccLabeledProjectionModel** and the supplied input image.

cfLabelProject

```
template<class M, class I, class R, class N>
void cfLabelProject (
  const ccLabeledProjectionModel<M> &model,
  const ccPelBuffer_const<I> &image,
  const ccPelBuffer<N> &normProjection,
  const ccPelBuffer<R> &rawProjection,
  bool preClearProj=true);
```

Computes the specified labeled projection, producing both normalized and non-normalized projection images.

Parameters

M, *I*, *R*, and *N* — Template parameters specifying the type of the projection model, source image, normalized projection image, and raw projection image. You must specify one of the following combinations of types:

M	I	R	N
c_UInt8	**c_UInt8**	**c_UInt32**	**c_UInt32**
c_UInt16	**c_UInt8**	**c_UInt16**	**c_UInt16**
c_UInt8	**c_UInt8**	**c_UInt32**	**c_UInt8**
c_UInt16	**c_UInt8**	**c_UInt16**	**c_UInt8**

model — The labeled projection model to use.

image — The input image.

normProjection — A reference to a **ccPelBuffer<N>** into which to place the normalized projection image.

rawProjection — A reference to a **ccPelBuffer<R>** into which to place the non-normalized projection image. The pixel type of *rawProjection* must be large enough to contain the product of the largest pixel value in *image* times the number of pixels in *model* with the most frequent label value.

In general, *rawProjection* should be of type **ccPelBuffer<c_UInt32>**.

preClearProj If *preClearProj* is true, this function will initialize the contents of *normProjection* and *rawProjection* to zero before projecting *image*.

If *preClearProj* is false, then this function will perform a raw projection of *image* into a temporary image, add the temporary image to *rawProjection*, then normalize the temporary image, then add the normalized image to *normProjection*.

cfLabelProjectNorm()

```
#include <ch_cvl/ch_cvl/lablproj.h>

cfLabelProjectNorm();
```

Global function to compute a normalized projection image using the supplied **ccLabeledProjectionModel** and the supplied input image.

cfLabelProjectNorm

```
template<class M, class I, class N>
void cfLabelProjectNorm (
  const ccLabeledProjectionModel<M> &model,
  const ccPelBuffer_const<I> &image,
  const ccPelBuffer<N> &normProjection, bool preClearProj);
```

Computes the specified labeled projection, producing a normalized projection image.

Parameters

M, *I*, and *N* — Template parameters specifying the type of the projection model, source image, and normalized projection image. You must specify one of the following combinations of types:

M	I	N
c_UInt8	**c_UInt8**	**c_UInt32**
c_UInt16	**c_UInt8**	**c_UInt16**

model — The labeled projection model to use.

image — The input image.

normProjection — A reference to a **ccPelBuffer<N>** into which to place the normalized projection image.

preClearProj — If *preClearProj* is true, this function will initialize the contents of *normProjection* to zero before projecting *image*.

If *preClearProj* is false, then this function will perform a raw projection of *image* into a temporary image, normalize the temporary image, then add the normalized image to *normProjection*.

Notes

The pixel type of *normProjection* must be large enough to contain the product of the largest pixel value in *image* times the number of pixels in *model* with the most frequent label value.

If you want to create a projection image of type **ccPelBuffer<c_UInt8>** but you expect intermediate values larger than 255, you should use the function **cfLabelProject()** and supply a second image with larger pixel values to store the raw pixel values.

Caution **cfLabelProjectNorm()** *does not throw an error if the projection values overflow the size of pixels in* normProjection.

cfLabelProjectRaw()

```
#include <ch_cvl/lablproj.h>
```

`cfLabelProjectRaw`();

Global function to compute a non-normalized projection image using the supplied **ccLabeledProjectionModel** and the supplied input image.

cfLabelProjectRaw

```
template<class M, class I, class R>
void cfLabelProjectRaw(
  const ccLabeledProjectionModel<M> &model,
  const ccPelBuffer_const<I> &image,
  const ccPelBuffer<R> &rawProjection,
  bool preClearProj);
```

Computes the specified labeled projection, producing a non-normalized projection image.

Parameters

M, *I*, and *R* — Template parameters specifying the type of the projection model, source image, and raw projection image. You must specify one of the following combinations of types:

M	I	R
c_UInt8	**c_UInt8**	**c_UInt32**
c_UInt16	**c_UInt8**	**c_UInt16**
c_UInt8	**c_UInt8**	**c_UInt32**

model — The labeled projection model to use.

image — The input image.

rawProjection — A reference to a **ccPelBuffer<N>** into which to place the non-normalized projection image. The pixel type of *rawProjection* must be large enough to contain the product of the largest pixel value in *image* times the number of pixels in *model* with the most frequent label value.

In general, *rawProjection* should be of type **ccPelBuffer<c_UInt32>**.

preClearProj — If *preClearProj* is true, this function will initialize the contents of *rawProjection* to zero before projecting *image*.

cfLabelProjectRaw()

cfLineFit()

```
#include <ch_cvl/circfit.h>

cfLineFit();
```

Global function to fit a line to a set of points.

cfLineFit

```
void cfLineFit(
  const ccLineFitParams &params,
  cmStd vector<cc2Vect> &pts,
  ccLineFitResults &results);
```

A global function that fits a line to the supplied set of points using the supplied **ccLineFitParams**.

Parameters

params — A **ccLineFitParams** describing the fitting method, number of outliers to ignore, and a maximum error threshold.

pts — A vector of points specified in a single coordinate system.

results — A reference to a **ccLineFitResults** into which the results will be placed. Any existing contents of *results* are cleared.

Throws

ccMathError::Singular
The number of points in *pts* is less than 2 after subtracting the number of points to ignore specified in *params*, or *pts* does not include two different points.

cfLineFit()

cfManualTrigger()

```
#include <ch_cvl/trigmodl.h>

cfManualTrigger();
```

Global function to retrieve a manual trigger model object reference.

cfManualTrigger

```
const ccTriggerModel& cfManualTrigger();
```

When you specify manual trigger as the trigger model, the application must call **ccAcqFifo::start()** to initiate each acquisition. Since no external input lines are involved, this trigger model is sometimes called a “software trigger.”

This is the default trigger model.

Notes

When **ccTriggerProp::triggerEnable()** is set to false, invocations of **ccAcqFifo::start()** cause acquisitions to queue. For acquisitions to proceed, **ccTriggerProp::triggerEnable()** must be set to true.

cfManualTrigger()

cfOCChangeCurrentKey()

```
#include <ch_cvl/oc.h>

cfOCChangeCurrentKey();
```

Global function to change the current key for a specified line position.

cfOCChangeCurrentKey

```
ccOCLineArrangementPtrh cfOCChangeCurrentKey(
  const ccOCLineArrangement& arr,
  c_Int32 lineIndex,
  c_Int32 posIndex,
  c_Int32 newKey);

ccOCLineArrangementPtrh cfOCChangeCurrentKey(
  const ccOCLineArrangement& arr,
  c_Int32 lineIndex,
  c_Int32 posIndex);
```

- ```
  ccOCLineArrangementPtrh cfOCChangeCurrentKey(
    const ccOCLineArrangement& arr,
    c_Int32 lineIndex,
    c_Int32 posIndex,
    c_Int32 newKey);
  ```

  Returns a new arrangement that is a copy of the supplied line arrangement, *arr*, with the exception that the current keys for the specified line and position are changed (replaced). The current key is directly replaced by the supplied *newKey*.

  Notes

  If current key replacement is made to change a character verified by an **ccOCVTool** object which has previously been trained with the line arrangement *arr*, it will be necessary to call the **retrain()** member function of that tool with the new arrangement returned by this function.

  Parameters

arr	A line arrangement.
lineIndex	The index of the line within the arrangement.
posIndex	The index of the position within the line for which to change the current key.
newKey	The new value for the current key.

  Throws
  ```

ccOCDefs::BadParams
The line or position index is not valid for the supplied arrangement.

ccOCDefs::BadKey
newkey is not in the key set for the specified position.

- ```
  ccOCLineArrangementPtrh cfOCChangeCurrentKey(
    const ccOCLineArrangement& arr,
    c_Int32 lineIndex,
    c_Int32 posIndex);
  ```

Returns a new arrangement that is a copy of the supplied line arrangement, *arr*, with the exception that the current keys for the specified line and position are changed (replaced). The current keys are directly replaced by a wildcard character.

Notes

If current key replacements are made to change characters verified by an **ccOCVTool** object which has previously been trained with the line arrangement *arr*, it will be necessary to call the **retrain()** member function of that tool with the new arrangement returned by this function.

Parameters

arr A line arrangement.

lineIndex The index of the line within the arrangement.

posIndex The index of the position within the line for which to change the current key.

Throws

ccOCDefs::BadParams
Either the line or position index is not valid for the supplied arrangement.
  ```

cfOCChangeCurrentKeys()

```
#include <ch_cvl/oc.h>

cfOCChangeCurrentKeys();
```

Global function to change the current keys for a specified line position.

cfOCChangeCurrentKeys

```
ccOCLineArrangementPtrh cfOCChangeCurrentKeys(
  const ccOCLineArrangement& arr,
  c_Int32 lineIndex,
  c_Int32 posIndex,
  const cmStd vector<c_Int32>& newKeys);

ccOCLineArrangementPtrh cfOCChangeCurrentKeys(
  const ccOCLineArrangement& arr,
  c_Int32 lineIndex,
  c_Int32 posIndex,
  const ccCvlString& newName);
```

- ```
  ccOCLineArrangementPtrh cfOCChangeCurrentKeys(
    const ccOCLineArrangement& arr,
    c_Int32 lineIndex,
    c_Int32 posIndex,
    const cmStd vector<c_Int32>& newKeys);
  ```

  Returns a new arrangement that is a copy of the supplied line arrangement, *arr*, with the exception that the current keys for the specified line and position are changed (replaced). The current keys are directly replaced by the supplied *newKeys*.

  Notes

  If current key replacements are made to change characters verified by an **ccOCVTool** object which has previously been trained with the line arrangement *arr*, it will be necessary to call the **retrain()** member function of that tool with the new arrangement returned by this function.

  Parameters

arr	A line arrangement.
lineIndex	The index of the line within the arrangement.
posIndex	The index of the position within the line for which to change the current key or keys.
newKeys	The new values for the current keys.
  ```

Throws

ccOCDefs::BadParams
The line or position index is not valid for the supplied arrangement.

ccOCDefs::BadKey
newKeys is not in the key set for the specified position.

- ```
  ccOCLineArrangementPtrh cfOCChangeCurrentKeys(
    const ccOCLineArrangement& arr,
    c_Int32 lineIndex,
    c_Int32 posIndex,
    const ccCvlString& newName);
  ```

Returns a new arrangement that is a copy of the supplied line arrangement, *arr*, with the exception that the current keys for the specified line and position are changed (replaced). The current keys are directly replaced by the supplied *newKeys*. The current keys are replaced by a single key that corresponds to the supplied character name.

Notes

If current key replacements are made to change characters verified by an **ccOCVTool** object which has previously been trained with the line arrangement *arr*, it will be necessary to call the **retrain()** member function of that tool with the new arrangement returned by this function.

Despite the name of this function, you cannot use it to specify more than a single current key.

Parameters

arr A line arrangement.

lineIndex The index of the line within the arrangement.

posIndex The index of the position within the line for which to change the current key.

newName The name of the character to select as the current key.

Throws

ccOCDefs::BadParams
Either the line or position index is not valid for the supplied arrangement.

ccOCDefs::BadKey
The key specified by *newName* does not correspond to a character in the alphabet for the line specified by the supplied line index.
  ```

cfOCRReclassifyAfterFielding()

```
#include <ch_cvl/ocrglue.h>

cfOCRReclassifyAfterFielding();
```

Global function to compute the classifier result according to the fielding.

cfOCRReclassifyAfterFielding

```
cmImport_cogocr void cfOCRReclassifyAfterFielding(
  const ccOCRDictionaryFielding &fielding,
  const ccOCRDictionaryResult& fieldingResult,
  const ccOCRClassifierLineResult &classifierResult,
  const ccOCSwapCharSet &swapCharSet,
  bool computeConfusionUsingOnlyAllowableCharacters,
  ccOCRClassifierLineResult &computedResult);
```

Computes the classifier result according to the fielding (based on the given *fieldingResult* and the given *classifierResult*).

Parameters

fielding The fielding.

fieldingResult The fielding result.

classifierResult The classifier result.

swapCharSet The swap character set.

computeConfusionUsingOnlyAllowableCharacters
If true, confusion is computed using only the allowable characters.

computedResult The computed result.

Notes

The output *computedResult* has the same number of character elements as the *fieldingResult resultString*, and they are in 1-1 correspondence.

The output *computedResult* may have a different number of character elements than the input *classifierResult*.

If characters are inserted in the fielding result, they will be reflected in the *computedResult* as position results with status *eFailed*, score 0, *characterCode eUnknown*.

If the confusion is not limited to only allowable characters, that is, *computeConfusionUsingOnlyAllowableCharacters* == false, then all the non-primary characters will be included as alternative characters (regardless of

whether they agree with the fielding). In other words, if *computeConfusionUsingOnlyAllowableCharacters* == false, then only the primary character is constrained to agree with the fielding.

If the fieldings vector is empty (**fielding.size()**==0), then the *classifierResult* is copied to the *computedResult.*

For each position, if the highest scoring character that satisfies the *allowableCharacterCodes* constraints does not satisfy the accept threshold, then the status is set to *eFailed*. Correspondingly, if the status is *eFailed*, then the primary character is unknown.

If *computeConfusionUsingOnlyAllowableCharacters* == false, then for each position, the confusion character is set to the highest scoring non-primary character; consequently, for each position, the *confusionCharacter* may be valid even if the primary character is unknown and the classifier status is failed.

For each position, if the primary character is the only matching character, the confusion character is set to a default constructed character.

Throws

ccOCRDictionaryDefs::BadParams

classifierResult.positionResults().size() != **fieldingResult.inputStringMulti().size()**

cfOCSegmentCharacters()

```
#include <ch_cvl/ocsegmnt.h>

cfOCSegmentCharacters();
```

Global function for segmenting.

cfOCSegmentCharacters

```
void cfOCSegmentCharacters(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccAffineRectangle& searchRegion,
  const ccOCCharSegmentRunParams& params,
  ccOCCharSegmentParagraphResult& result,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);

void cfOCSegmentCharacters(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccOCCharSegmentRunParams& params,
  ccOCCharSegmentParagraphResult& result,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);

void cfOCSegmentCharacters(
  const ccPelBuffer_const<c_UInt8>& image,
  const cmStd vector<ccAffineRectangle>& searchRegions,
  const ccOCCharSegmentRunParams& params,
  ccOCCharSegmentResult& result,
  ccDiagObject* diagobj=0,
  c_UInt32 diagFlags=0);
```

-
```
void cfOCSegmentCharacters(
  const ccPelBuffer_const<c_UInt8>& image,
  const ccAffineRectangle& searchRegion,
  const ccOCCharSegmentRunParams& params,
```

```
ccOCCharSegmentParagraphResult& result,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);
```

Segments one line of characters contained in the image within the region of interest (ROI) specified by *searchRegion*.

The ROI should contain one line of characters and surrounding featureless, but possibly nonuniform and/or noisy background; there should be no strong features in the ROI other than the one line of characters.

The ROI is specified in the client coordinates of the input image. The x-axis should be approximately parallel to the baseline of the contained line of characters, with its positive direction pointing along the reading direction.

The skew should be the approximate value of the skew of the line of characters.

If **params.angleHalfRange()** is greater than zero, the angle of the line is refined. If **params.skewHalfRange()** is greater than zero, the skew of the line is refined.

Even if no characters are found, the paragraph result contains exactly one line result.

Parameters

image	The input image for the OCR Segmenter tool.
searchRegion	The ROI.
params	The set of input parameters.
result	The result of the segmentation.
diagobj	An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.
diagFlags	The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values: *ccDiagDefs::eInputs* *ccDiagDefs::eIntermediate* *ccDiagDefs::eResults* with one of the following values: *ccDiagDefs::eRecordOn* *ccDiagDefs::eRecordOff*

Throws

ccOCRDefs::UnboundWindow
image is unbound.

ccOCRDefs::NotImplemented
image.imageFromClientXformBase()->**isLinear()** returns false, or
searchRegion is partially outside the input image.

ccOCRDefs::BadParams
searchRegion is degenerate, or
searchRegion is entirely outside the input image, or
any of the following is true:

- **params.useCharacterMaxWidth()** == true && **params.characterMinWidth()** > **params.characterMaxWidth()**
- **params.useCharacterMaxHeight()** == true && **params.characterMinHeight()** > **params.characterMaxHeight()**
- **params.spaceParams().spaceMinWidth()** > **params.spaceParams().spaceMaxWidth()**
- *searchRect* corresponds to less than 1 pel in either x or y.

- ```
  void cfOCSegmentCharacters(
    const ccPelBuffer_const<c_UInt8>& image,
    const ccOCCharSegmentRunParams& params,
    ccOCCharSegmentParagraphResult& result,
    ccDiagObject* diagobj=0,
    c_UInt32 diagFlags=0);
  ```

  Segments one line of characters contained in the image.

  The ROI is the entire image window.

  The ROI should contain one line of characters and surrounding featureless, but possibly nonuniform and/or noisy background; there should be no strong features in the ROI other than the one line of characters.

  The contained line of characters should have its baseline approximately horizontal in the image, with the reading direction being from left to right.

  If **params.angleHalfRange()** is greater than zero, the angle of the line is refined. If **params.skewHalfRange()** is greater than zero, the skew of the line is refined.

  Even if no characters are found, the paragraph result contains exactly one line result.

  Parameters

image	The input image for the OCR Segmenter tool.
params	The set of input parameters.
result	The result of the segmentation.
  ```

diagobj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccOCRDefs::UnboundWindow
image is unbound.

ccOCRDefs::NotImplemented
image.imageFromClientXformBase()->**isLinear()** returns false.

ccOCRDefs::BadParams
Any of the following is true:

- **params.useCharacterMaxWidth()** == true && **params.characterMinWidth()** > **params.characterMaxWidth()**
- **params.useCharacterMaxHeight()** == true && **params.characterMinHeight()** > **params.characterMaxHeight()**
- **params.spaceParams().spaceMinWidth()** > **params.spaceParams().spaceMaxWidth()**
- *image* is smaller than 1 pel in either x or y.

- ```
  void cfOCSegmentCharacters(
    const ccPelBuffer_const<c_UInt8>& image,
    const cmStd vector<ccAffineRectangle>& searchRegions,
    const ccOCCharSegmentRunParams& params,
  ```
  ```

```
ccOCCharSegmentResult& result,
ccDiagObject* diagobj=0,
c_UInt32 diagFlags=0);
```

Segments one or more paragraphs of characters contained in the image within regions of interest specified by *searchRegions*.

Each ROI is segmented as described in the single-ROI overload of **cfOCSegmentCharacters()** above.

result contains one paragraph result for each ROI, even if no characters are found in a given ROI.

Each ROI is treated completely independently. For example, lines in different ROIs may have different angles and/or skews.

Parameters

image The input image for the OCR Segmenter tool.

searchRegions The ROIs.

params The set of input parameters.

result The result of the segmentation.

diagobj An optional **ccDiagObject**. If you supply a value for *diagobj*, then the tool will record diagnostic information in the supplied object.

diagFlags The optional diagnostic flags. *diagFlags* must be composed by ORing together one or more of the following values:

ccDiagDefs::eInputs
ccDiagDefs::eIntermediate
ccDiagDefs::eResults

with one of the following values:

ccDiagDefs::eRecordOn
ccDiagDefs::eRecordOff

Throws

ccOCRDefs::UnboundWindow
image is unbound.

ccOCRDefs::BadParams
Any of the following is true:

- The vector *searchRegions* is empty.
- Any element of *searchRegions* is degenerate.

- Any element of *searchRegions* is entirely outside the input image.
- **params.useCharacterMaxWidth()** == true && **params.characterMinWidth()** > **params.characterMaxWidth()**
- **params.useCharacterMaxHeight()** == true && **params.characterMinHeight()** > **params.characterMaxHeight()**
- **params.spaceParams().spaceMinWidth()** > **params.spaceParams().spaceMaxWidth()**
- Any of the *searchRects* correspond to less than 1 pel in either x or y.

ccOCRDefs::NotImplemented

image.imageFromClientXformBase()->**isLinear()** returns false, or
any element of *searchRegions* is partially outside the input image.

cfPDF417Decode()

```
#include <ch_cvl/pdf417.h>

cfPDF417Decode();
```

Global function to find and decode a PDF417 stacked barcode symbol.

cfPDF417Decode

```
void cfPDF417Decode(
  const ccPelBuffer_const<c_UInt8>& image,
  ccPDF417Result &result);
```

- ```
  void cfPDF417Decode(
    const ccPelBuffer_const<c_UInt8>& image,
    ccPDF417Result &result);
  ```

  Finds the PDF417 symbol in a given image and decodes it. Returns a result via the result object. The PDF417 symbol tool is able to detect the encoded information directly from the image. Therefore, no learning phase is needed before decoding.

  Parameters

image	Image that contains the PDF417 barcode symbol. The image must have at least quiet zones 4X wide on each side of the symbol, where X is the width of the narrowest bar.
result	Object in which to store the result.

  Throws

  ccPDF417Defs::BadImage
  Image is unbound.
  ```

cfPDF417Decode()

cfPelAdd()

```
#include <ch_cvl/pelfunc.h>

cfPelAdd();
```

Global function to add the pixels of two windows.

Note When this function computes the greatest common region of two images it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

cfPelAdd

```
template<class P>
ccPelBuffer<P> cfPelAdd(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);

template<class P>
ccPelBuffer<P> cfPelAdd(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2, c_Int32 output_mode);

template<class S1, class S2, class D>
void cfPelAdd(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);

template<class S1, class S2, class D>
void cfPelAdd(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2,
  ccPelBuffer<D>& dest, c_Int32 output_mode);
```

-
```
template<class P>
ccPelBuffer<P> cfPelAdd(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);
```

Adds the pixels in the greatest common region of *src1* and *src2* using the default output mode (*ccPelFunc::eAddDefault*), writes the resulting pixels values to a new root image, and returns a window that is bound to that image.

Parameters

P	Template parameter specifying the type of the windows whose pixels are being added and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
src1	A bound window.
src2	A bound window.

Throws

ccPel::UnboundWindow
src1 or *src2* is unbound.

- ```
  template<class P>
  ccPelBuffer<P> cfPelAdd(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2, c_Int32 output_mode);
  ```

  Applies the user-specified method of handling overflow as it adds the pixels in the greatest common region of *src1* and *src2*, writes the resulting pixel values to a new root image, and returns a window that is bound to that image.

  Parameters

  P — Template parameter specifying the type of the windows whose pixels are being added and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  src1 — A bound window.

  src2 — A bound window.

  output_mode — The method to use to handle overflow. *output_mode* must be one of the following values:

  ccPelFunc::eAddDefault
  ccPelFunc::eAddAbs
  ccPelFunc::eAddClamp
  ccPelFunc::eAddShft
  ccPelFunc::eAddLimit

  See **add_mode** on page 2384.

  Throws

  ccPel::UnboundWindow
  src1 or *src2* is unbound.

  ccPelFunc::BadParams
  output_mode is not a valid addition mode.

- ```
  template<class S1, class S2, class D>
  void cfPelAdd(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
  ```

  Adds the pixels in the region of *src1* and *src2* defined by greatest common region of *src1* and *src2* and *dest* using the default output mode (*ccPelFunc::eAddDefault*), and writes the resulting pixel values to *dest*.

Parameters

S1, S2, D Template parameters that specify the type of the two source windows and the destination window. The following combinations of types are permitted for *S1, S2,* and *D:*

S1	S2	*D*
c_UInt8	c_UInt8	c_UInt8
c_UInt16	c_UInt16	c_UInt16
c_UInt32	c_UInt32	c_UInt32
c_UInt8	c_UInt16	c_UInt16
c_UInt8	c_UInt32	c_UInt32

src1 A bound window.

src2 A bound window.

dest A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, and *dest* will be added and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1* and *src2*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then added and the resulting values are written to *dest*.

Throws

ccPel::UnboundWindow

src1 or *src2* is unbound.

- ```
  template<class S1, class S2, class D>
  void cfPelAdd(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2,
    ccPelBuffer<D>& dest, c_Int32 output_mode);
  ```

  Applies the user-specified method of handling overflow as it adds the pixels in the region of *src1* and *src2* defined by greatest common region of *src1* and *src2* and *dest,* and writes the resulting pixels values to *dest*.
  ```

Parameters

S1, S2, D

Template parameters that specify the type of the two source windows and the destination window. The following combinations of types are permitted for *S1, S2,* and *D:*

S1	S2	*D*
c_UInt8	**c_UInt8**	**c_UInt8**
c_UInt16	**c_UInt16**	**c_UInt16**
c_UInt32	**c_UInt32**	**c_UInt32**
c_UInt8	**c_UInt16**	**c_UInt16**

src1

A bound window.

src2

A bound window.

dest

A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, and *dest* will be added and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1* and *src2*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then added and the resulting values are written to *dest*.

output_mode

The method to use to handle overflow. *output_mode* must be one of the following values:

ccPelFunc::eAddDefault
ccPelFunc:: eAddAbs
ccPelFunc::eAddClamp
ccPelFunc::eAddShft
ccPelFunc::eAddLimit

See **add_mode** on page 2384.

Throws

ccPel::UnboundWindow

src1 or *src2* is unbound.

ccPelFunc::BadParams

output_mode is not a valid addition mode.

cfPelClear()

```
#include <ch_cvl/pelfunc.h>

cfPelClear();
```

Global function to set all pixels of a window to zero.

P Template parameter specifying the type of the pixels in *win*. *P* must be one of **c_UInt8**, **c_UInt16**, **c_UInt32**, **double**, **ccPackedRGB16Pel**, **ccPackedRGB32Pel**.

cfPelClear

```
template<class P> void cfPelClear(
  const ccPelBuffer<P>& win);
```

Clears a window by setting its pixels to zero.

Parameters

win A bound window.

Throws

ccPel::UnboundWindow
win is unbound.

cfPelClear()

cfPelCopy()

```
#include <ch_cvl/pelfunc.h>

cfPelCopy();
```

Global function to copy the pixels in a window.

cfPelCopy

```
template<class P>
ccPelBuffer<P> cfPelCopy(const ccPelBuffer_const<P>& src);

template<class S, class D>
void cfPelCopy(const ccPelBuffer_const<S>& src,
 ccPelBuffer<D>& dest);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelCopy(const ccPelBuffer_const<P>& src);
  ```

  Copies the pixels in *src* to a new root image and returns a window that is bound to that image.

  Parameters

  P — Template parameter specifying the type of the window to be returned by the function and of the pixels in *src*. *P* must be one of **c_UInt8**, **c_UInt16**, **c_UInt32** or **double**.

  src — A bound window.

  Throws

  ccPel::UnboundWindow — *src* is unbound.

- ```
  template<class S, class D>
  void cfPelCopy(const ccPelBuffer_const<S>& src,
   ccPelBuffer<D>& dest);
  ```

  Copies the pixels in *src* to *dest*.

  Parameters

  S — Template parameter specifying the type of the pixels in *src*. *S* must be one of **c_UInt8**, **c_UInt16**, **c_UInt32**, **ccPackedRGB16Pel**, or **ccPackedRGB32Pel**.

D Template parameter specifying the type of the pixels in *dest*. *D* must be either the same type or a larger type than *S*.

If *S* is	*D* must be one of
c_UInt8	**c_UInt8**, **c_UInt16**, or **c_UInt32**
c_UInt16	**c_UInt16** or **c_UInt32**
c_UInt32	**c_UInt32**
ccPackedRGB16Pel	**ccPackedRGB16Pel**
ccPackedRGB32Pel	**ccPackedRGB32Pel**
cc3PlanePelBuffer	**cc3PlanePelBuffer**

src A bound window.

dest A bound or unbound window. If *dest* is bound, the pixels in the region of *src* determined by the greatest common region of *src* and *dest* are copied. If *dest* is unbound, a root image for *dest* is allocated and all the pixels in *src* are copied to *dest*.

Throws

ccPel::UnboundWindow
src is unbound.

Notes

When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

cfPelDivideByVal()

```
#include <ch_cvl/pelfunc.h>

cfPelDivideByVal();
```

Global function to divide the pixels in a window by a specified value.

cfPelDivideByVal

```
template<class P, class Div>
ccPelBuffer<P> cfPelDivideByVal(
  const ccPelBuffer_const<P>& src, Div divisor);

template<class S, class D, class Div>
void cfPelDivideByVal(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest, Div divisor);
```

- ```
  template<class P, class Div>
  ccPelBuffer<P> cfPelDivideByVal(
    const ccPelBuffer_const<P>& src, Div divisor);
  ```

  Divides the pixels in *src* by the specified *divisor*, writes the resulting pixel values to a new root image, and returns a window that is bound to that image.

  Parameters

  P — Template parameter specifying the type of the windows whose pixels are being divided and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  Div — Template parameter specifying the type of the divisor. *Div* must always be **double**.

  src — A bound window.

  divisor — The nonzero value by which the pixels in *src* are divided.

  Throws

  ccPel::UnboundWindow — *src* is unbound.

- ```
  template<class S, class D, class Div>
  void cfPelDivideByVal(const ccPelBuffer_const<S>& src,
    ccPelBuffer<D>& dest, Div divisor);
  ```

  In the greatest common region of *src* and *dest*, divides the pixels in *src* by the value of *divisor* and writes the resulting pixel values to *dest*.

Parameters

S Template parameter specifying the type of the windows whose pixels are being divided and of the window that the function returns. *S* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

D Template parameter specifying the type of the pixels in *dest*. *D* must be either the same type or a smaller type than *S*.

If *S* is	***D* must be one of**
c_UInt8	**c_UInt8**
c_UInt16	**c_UInt8** or **c_UInt16**
c_UInt32	**c_UInt8**, **c_UInt16**, **c_UInt32**

Div Template parameter specifying the type of the divisor. *Div* must always be **double**.

src A bound window.

dest A bound or unbound window. If *dest* is bound, the pixels in the region of *src* determined by the greatest common region of *src* and *dest* are divided by *divisor*. If *dest* is unbound, a root image for *dest* is allocated, the pixels in *src* are divided, and the resulting pixel values are written to *dest*.

divisor A nonzero value by which the pixels in *src* are divided.

Throws

ccPel::UnboundWindow
src is unbound.

cfPelEqual()

```
#include <ch_cvl/pelfunc.h>

cfPelEqual();
```

Global function to compare the pixels in two windows.

cfPelEqual

```
template<class P1, class P2>
bool cfPelEqual(const ccPelBuffer_const<P1>& win1,
  const ccPelBuffer_const<P2>& win2);
```

Compares the pixels in the greatest common region of *win1* and *win2* and returns true if all the pixel values in the two windows are equal; otherwise the function returns false.

Parameters

P1, *P2*	Template parameters specifying the type of the pixels in *win1* and *win2*. *P1* and *P2* are always the same, and must be one of **c_UInt8**, **c_Int8**, **c_Int16**, **c_UInt16**, **c_UInt32** or **double**.
win1	A bound window.
win2	A bound window.

Notes

When this function computes the greatest common region of *win1* and *win2* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

If the greatest common region is of 0 size, that is, contains no pixels, **cfPelEqual()** returns true.

Throws

ccPel::UnboundWindow
win1 or *win2* is unbound.

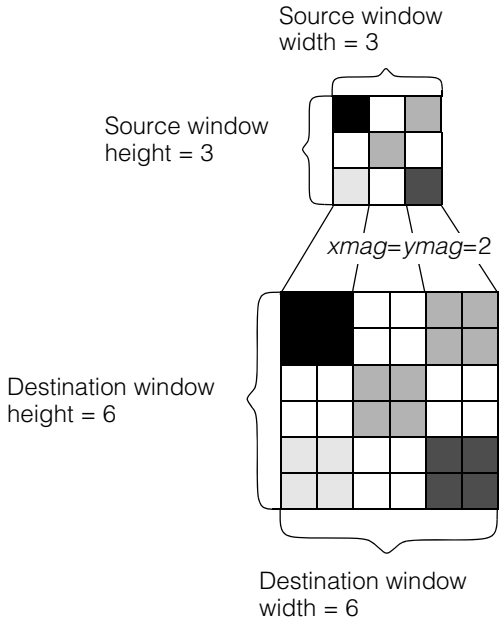
cfPelEqual()

cfPelExpand()

```
#include <ch_cvl/pelfunc.h>

cfPelExpand();
```

Global function to copy each pixel in the source window to a rectangular block of pixels in the destination window, enlarging the source image. The following figure shows an example of expansion.



The destination image is divided into blocks of pixels, starting at the upper left pixel, with *xmag* and *ymag* defining the size of the blocks. Then each block in the destination image is filled in with copies of the source pixel corresponding to the destination pixel block as shown above.

cfPelExpand()

cfPelExpand

```
template<class P>
ccPelBuffer<P> cfPelExpand(
  const ccPelBuffer_const<P>& src,
  c_Int32 xmag, c_Int32 ymag);

template<class S, class D>
void cfPelExpand(const ccPelBuffer_const<S>& src,
ccPelBuffer<D>& dest, c_Int32 xmag, c_Int32 ymag);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelExpand(
    const ccPelBuffer_const<P>& src,
    c_Int32 xmag, c_Int32 ymag);
  ```

  Expands the pixels in *src* by the values of *xmag* by *ymag*, writes the resulting pixels to a new root image, and returns a window that is bound to that image.

  Parameters

P	Template parameter specifying the type of the window whose pixels are being magnified and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**, **ccPackedRGB16Pel**, or **ccPackedRGB32Pel**.
src	A bound window.
xmag, ymag	The values by which source pixels are magnified in the x and y dimensions. *xmag* and *ymag* must be greater than 0.

  Throws

  ccPel::UnboundWindow
  src is unbound.

- ```
  template<class S, class D>
  void cfPelExpand(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest, c_Int32 xmag, c_Int32 ymag);
  ```

  Expands the pixels in *src* by the values of *xmag* by *ymag* and writes the resulting pixels to *dest*.

  Parameters

S, *D*	Template parameters specifying the type of the source and destination windows. *S* and *D* are always the same, and must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**, **ccPackedRGB16Pel**, or **ccPackedRGB32Pel**.
src	A bound window.
dest	A bound or unbound window.

If *dest* is unbound, a new root image is allocated to accommodate the results of the expansion, and *dest* is bound to the root image. All the pixels in *src* are then expanded and the results are written to *dest*.

If *dest* is bound, the function calculates the size that *src* would be after expansion, then determines the greatest common region of this expanded *src* with *dest* to determine how much of *src* is to be expanded.

When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

xmag, ymag — The values by which source pixels are magnified in the x and y dimensions. *xmag* and *ymag* must be greater than 0.

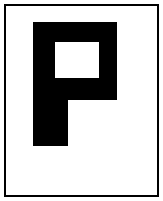
cfPelExpand()

cfPelFlipH()

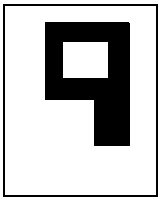
```
#include <ch_cvl/pelfunc.h>

cfPelFlipH();
```

Global function to flip the pixels in a window horizontally. The following figure shows an example of a horizontal flip.



Source window



Window after horizontal flip

cfPelFlipH

```
template<class P>
ccPelBuffer<P> cfPelFlipH(
  const ccPelBuffer_const<P>& src);

template<class S, class D>
void cfPelFlipH(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelFlipH(
    const ccPelBuffer_const<P>& src);
  ```

  Flips the pixels in *src* horizontally, copies them to a new root image, and returns a window that is bound to that image.

  Parameters

  P — Template parameter specifying the type of the window to be returned by the function and of the pixels in *src*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  src — A bound window.

  Throws

  ccPel::UnboundWindow — *src* is unbound.
  ```

- ```
  template<class S, class D>
  void cfPelFlipH(const ccPelBuffer_const<S>& src,
    ccPelBuffer<D>& dest);
  ```

  Flips the pixels in *src* and copies them to *dest*. If the same window is specified for *src* and *dest*, the operation is performed in place.

  Parameters

  S — Template parameter specifying the type of the pixels in *src*. *S* must be one of **c_UInt8**, **c_UInt16** or **c_UInt32**.

  D — Template parameter specifying the type of the pixels in *dest*. *D* must be the same type as *S*.

  src — A bound window.

  dest — A bound or unbound window. If *dest* is bound, the pixels in the region of *src* determined by the greatest common region of *src* and *dest* are flipped and copied. If *dest* is unbound, a root image for *dest* is allocated and all the pixels in *src* are flipped and copied to *dest*.

  When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

  Throws

  ccPel::UnboundWindow
  src is unbound.

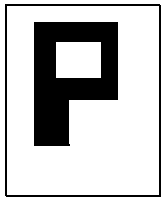
  ccPelFunc::Overlap
  src and *dest* overlap, but are not identical.
  ```

cfPelFlipV()

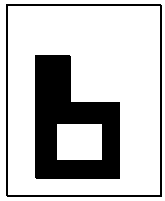
```
#include <ch_cvl/pelfunc.h>

cfPelFlipV();
```

Global function to flip the pixels in a window vertically. The following figure shows an example of a vertical flip.



Source window



Window after vertical flip

cfPelFlipV

```
template<class P>
ccPelBuffer<P> cfPelFlipV(
  const ccPelBuffer_const<P>& src);

template<class S, class D>
void cfPelFlipV(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelFlipV(
    const ccPelBuffer_const<P>& src);
  ```

  Flips the pixels in *src* vertically, copies them to a new root image, and returns a window that is bound to that image.

  Parameters

  P — Template parameter specifying the type of the window to be returned by the function and of the pixels in *src*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  src — A bound window.

  Throws

  ccPel::UnboundWindow — *src* is unbound.
  ```

- ```
  template<class S, class D>
  void cfPelFlipV(const ccPelBuffer_const<S>& src,
    ccPelBuffer<D>& dest);
  ```

  Flips the pixels in *src* and copies them to *dest*. If the same window is specified for *src* and *dest*, the operation is performed in place.

  Parameters

  S — Template parameter specifying the type of the pixels in *src*. *S* must be **c_UInt8**, **c_UInt16** or **c_UInt32**.

  D — Template parameter specifying the type of the pixels in *dest*. *D* must be the same type as *S*.

  src — A bound window.

  dest — A bound or unbound window. If *dest* is bound, the pixels in the region of *src* determined by the greatest common region of *src* and *dest* are flipped and copied. If *dest* is unbound, a root image for *dest* is allocated and all the pixels in *src* are flipped and copied to *dest*.

  When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

  Throws

  ccPel::UnboundWindow
  src is unbound.

  ccPelFunc::Overlap
  src and *dest* overlap, but are not identical.
  ```

cfPelHistogram()

```
#include <ch_cvl/histo.h>

cfPelHistogram();
```

Global function to compute the histogram of an input image.

cfPelHistogram

```
template<class P>
void cfPelHistogram(const ccPelBuffer_const<P>& image,
 cmStd vector<c_UInt32>& histogram);

template<class P>
void cfPelHistogram(const ccPelBuffer_const<P>& image,
 cmStd vector<c_UInt32>& histogram, bool& flag);
```

- ```
  template<class P>
  void cfPelHistogram(const ccPelBuffer_const<P>& image,
   cmStd vector<c_UInt32>& histogram);
  ```

  For each pixel in *image*, the value of the element of *histogram* with an index equal to the pixel's value is incremented. If *histogram* has a size of 0, **cfPelHistogram()** expands *histogram* so that it has an element at every index found in *image*. If *histogram* has a nonzero size, **cfPelHistogram()** does not perform any bounds checking as it computes the histogram.

  Note *histogram* is not cleared or initialized by this function.

  Caution *If you supply a nonzero size histogram to this function, you must ensure that it contains an element that corresponds to every value found in* image*. If it does not, your program will experience run-time memory corruption.*

  Parameters

  image The image for which to compute a histogram. *image* must be of type **c_UInt8** or **c_UInt16**.

  histogram An array of **c_UInt32** into which the histogram is written. If *histogram* has no elements, it is resized so that all values in *image* can be recorded in it. If *histogram* has any elements then it is not resized and no bounds checking is performed.

  Throws

  ccPel::UnboundWindow
  image is not bound.
  ```

- ```
  template<class P>
  void cfPelHistogram(const ccPelBuffer_const<P>& image,
    cmStd vector<c_UInt32>& histogram, bool& flag);
  ```

  For each pixel in *image*, the value of the element of *histogram* with an index equal to the pixel's value is incremented. If *histogram* does not contain an element at an index that corresponds to a pixel in *image*, the element of *histogram* closest to the value is incremented and *flag* is set to true.

Note *histogram* is not cleared or initialized by this function.

Caution *If you supply a nonzero size histogram to this function, you must ensure that it contains an element that corresponds to every value found in image. if it does not, your program will experience run-time memory corruption.*

Parameters

image The image for which to compute a histogram. *image* must be of type **c_UInt8**.

histogram An array of **c_UInt32** into which the histogram is written. *histogram* must have at least one element.

flag A reference to a **bool**. The value referred to by *flag* is set to true if clamping occurred.

Throws

ccPel::UnboundWindow
image is not bound.
  ```

cfPelMap()

```
#include <ch_cvl/pelmap.h>

cfPelMap();
```

Global function to map the pixels in a **ccPelBuffer** according to a table.

cfPelMap

```
template<class P>
ccPelBuffer<P> cfPelMap(
  const ccPelBuffer_const<P>& src,
  const cmStd vector<P>& pmap);

template<class S, class D>
void cfPelMap(
  const ccPelBuffer_const<S>& src,
  const cmStd vector<D>& pmap,
  ccPelBuffer<D>& dest);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelMap(
    const ccPelBuffer_const<P>& src,
    const cmStd vector<P>& pmap);
  ```

  Returns a new pel buffer that is a result of mapping each pixel in *src* to the corresponding value in *pmap*. Each pixel value in *src* is used as an index into *pmap*. The pixel value at that offset is used as the output pixel value.

  Parameters

P	Template parameter specifying the type of the pel buffer whose pixels are being mapped and of the pel buffer that the function returns. *P* must be **c_UInt8**.
src	The source pel buffer.
pmap	The mapping table. This is a vector of all the possible values in src. For an 8-bit image, for example, *pmap* must contain at least 256 elements.

  Throws

  ccPel::UnboundWindow
  src is not bound.
  ```

- ```
  template<class S, class D>
  void cfPelMap(
    const ccPelBuffer_const<S>& src,
    const cmStd vector<D>& pmap,
    ccPelBuffer<D>& dest);
  ```

  Returns in *dest* the result of mapping each pixel in *src* to the corresponding value in *pmap*. Each pixel value in *src* is used as an index into *pmap*. The pixel value at that offset is used as the output pixel value.

  Parameters

P	Template parameter specifying the type of the pel buffer whose pixels are being mapped. *P* must be **c_UInt8**.
D	Template parameter specifying the type of the pel buffer used to hold the results. *D* must be the same type as *P*.
src	The source pel buffer.
pmap	The mapping table. This is a vector of all the possible values in src. For an 8-bit image, for example, *pmap* must contain at least 256 elements.
dest	The destination pel buffer. If *dest* is unbound, a pel buffer of the same dimensions as *src* is allocated.

  Throws

  ccPel::UnboundWindow
  : *src* is not bound.

  ccPelFunc::Overlap
  : *src* and *dest* overlap.
  ```

cfPelMax()

```
#include <ch_cvl/pelfunc.h>

cfPelMax();
```

Global function to find the maximum pixel value in a window, or to compute an image where each pixel is the maximum of the values of the corresponding pixels in two input images.

cfPelMax

```
template<class P> P cfPelMax(
  const ccPelBuffer_const<P>& win);

template<class P> ccPelBuffer<P> cfPelMax(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);

template<class S1, class S2, class D> void cfPelMax(
  const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
```

- ```
  template<class P> P cfPelMax(
    const ccPelBuffer_const<P>& win);
  ```

  Returns the maximum pixel value in *win*.

  Parameters

  P — Template parameter specifying the type of the function's return value and of the pixels in *win*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  win — A bound window.

  Throws

  ccPel::UnboundWindow — *win* is unbound.

- ```
  template<class P> ccPelBuffer<P> cfPelMax(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2);
  ```

  Returns an image where each pixel is the maximum of the values of the corresponding pixels in *src1* and *src2*.

Parameters

P Template parameter specifying the type of pixels in *src1*, *src2*, and the returned window. *P* must be **c_UInt8**.

src1 A bound window.

src2 A bound window.

Throws

ccPel::UnboundWindow
src1 or src2 is unbound.

- ```
  template<class S1, class S2, class D> void cfPelMax(
    const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
  ```

Computes an image where each pixel is the maximum of the values of the corresponding pixels in *src1* and *src2* and places the image in *dest*.

Parameters

S1 Template parameter specifying the type of pixels in *src1*. *S1* must be **c_UInt8**.

S2 Template parameter specifying the type of pixels in *src2*. *S2* must be **c_UInt8**.

D Template parameter specifying the type of pixel in the destination window. *D* must be **c_UInt8**.

src1 A bound window.

src2 A bound window.

dest A window into which the result is placed. If *dest* is initially unbound, it is allocated to the same dimension as the greatest common region of *src1* and *src2*.

Throws

ccPel::UnboundWindow
src1 or src2 is unbound.

ccPelFunc::Overlap
src1 or src2 overlaps *dest* but they are not identical (after adjusting for the greatest common region).
  ```

cfPelMedian3x3()

```
#include <ch_cvl/pelfunc.h>

cfPelMedian3x3();
```

Global function that writes to a destination window the median value of each 3x3 neighborhood of pixels in the source window.

Pixels that are not at the center of a 3x3 neighborhood are not processed. This means that for a *src* window with a size of *width* x *height* pixels, the function returns a window that is, at most, (*width* - 2) x (*height* - 2) pixels.

cfPelMedian3x3

```
ccPelBuffer<c_UInt8> cfMedian3x3(
  const ccPelBuffer_const<c_UInt8>& src);

void cfPelMedian3x3(const ccPelBuffer_const<c_UInt8>& src,
  ccPelBuffer<c_UInt8>& dest);
```

- ```
  ccPelBuffer<c_UInt8> cfMedian3x3(
    const ccPelBuffer_const<c_UInt8>& src);
  ```

  Performs a 3x3 median filtering of the pixels in *src*, writes each resulting pixel value to a new root image, and returns a window that is bound to that image.

  Parameters

  src A bound window.

  Throws

  ccPel::UnboundWindow
  src is unbound.

- ```
  void cfPelMedian3x3(const ccPelBuffer_const<c_UInt8>& src,
    ccPelBuffer<c_UInt8>& dest);
  ```

  Performs a 3x3 median filtering of the pixels in *src* and writes each resulting pixel value to *dest*.

  Parameters

  src A bound window.

  dest A bound or unbound window.

  If *dest* is bound, the function determines the greatest common region of *src* with *dest* to determine how much of *src* is to be filtered. In calculating the greatest common region, the function

excludes the top and bottom rows and the left and right columns of pixels in *src* because they are not at the center of a 3x3 neighborhood.

If *dest* is unbound, a new root image is allocated to accommodate the results of the median filtering, and *dest* is bound to the root image. All the 3x3 neighborhoods in *src* are then filtered and the results are written to *dest*.

cfPelMin()

```
#include <ch_cvl/pelfunc.h>

cfPelMin();
```

Global function to find the minimum pixel value in a window, or to compute an image where each pixel is the minimum of the values of the corresponding pixels in two input images.

cfPelMin

```
template<class P> P cfPelMin(
  const ccPelBuffer_const<P>& win);

template<class P> ccPelBuffer<P> cfPelMin(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);

template<class S1, class S2, class D> void cfPelMin(
  const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
```

- ```
  template<class P> P cfPelMin(
    const ccPelBuffer_const<P>& win);
  ```

  Returns the minimum pixel value in *win*.

  Parameters

  P — Template parameter specifying the type of the function's return value and of the pixels in *win*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  win — A bound window.

  Throws

  ccPel::UnboundWindow — *win* is unbound.

- ```
  template<class P> ccPelBuffer<P> cfPelMin(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2);
  ```

  Returns an image where each pixel is the minimum of the values of the corresponding pixels in *src1* and *src2*.

Parameters

P Template parameter specifying the type of pixels in *src1*, *src2*, and the returned window. *P* must be **c_UInt8**.

src1 A bound window.

src2 A bound window.

Throws

ccPel::UnboundWindow
src1 or src2 is unbound.

- ```
  template<class S1, class S2, class D> void cfPelMin(
    const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
  ```

Computes an image where each pixel is the minimum of the values of the corresponding pixels in *src1* and *src2* and places the image in *dest*.

Parameters

S1 Template parameter specifying the type of pixels in *src1*. *S1* must be **c_UInt8**.

S2 Template parameter specifying the type of pixels in *src2*. *S2* must be **c_UInt8**.

D Template parameter specifying the type of pixel in *dest*. *D* must be **c_UInt8**.

src1 A bound window.

src2 A bound window.

dest A window into which the result is placed. If *dest* is initially unbound, it is allocated to the same dimension as the greatest common region of *src1* and *src2*.

Throws

ccPel::UnboundWindow
src1 or src2 is unbound.

ccPelFunc::Overlap
src1 or src2 overlaps *dest* but they are not identical (after adjusting for the greatest common region).
  ```

cfPelMinmax()

```
#include <ch_cvl/pelfunc.h>

cfPelMinmax();
```

Global function to find the minimum and maximum pixel values in a window.

cfPelMinmax

```
template<class P> void cfPelMinmax(
  const ccPelBuffer_const<P>& win, P& min, P& max);
```

Returns the minimum and maximum pixel values in *win*.

Parameters

P	Template parameter specifying the type of the pixels in *win* and of the *min* and *max* output parameters. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
win	A bound window.
min	A reference to an integer that will hold the window's minimum pixel value.
max	A reference to an integer that will hold the window's maximum pixel value.

Throws

ccPel::UnboundWindow
win is unbound.

cfPelMinmax()

cfPelMult()

```
#include <ch_cvl/pelfunc.h>

cfPelMult();
```

Global function to multiply the pixels of two windows.

Note When this function computes the greatest common region of two images it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

cfPelMult

```
template<class P>
ccPelBuffer<P> cfPelMult(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);

template<class P>
ccPelBuffer<P> cfPelMult(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2, c_Int32 output_mode);

template<class S1, class S2, class D>
void cfPelMult(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);

template<class S1, class S2, class D>
void cfPelMult(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2,
  ccPelBuffer<D>& dest, c_Int32 output_mode);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelMult(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2);
  ```

  Multiplies the pixels in the greatest common region of *src1* and *src2* using the default output mode (*ccPelFunc::eMultDefault*), writes the resulting pixels values to a new root image, and returns a window that is bound to that image.

  Parameters

P	Template parameter specifying the type of the windows whose pixels are being multiplied and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
src1	A bound window.
src2	A bound window.
  ```

Throws

ccPel::UnboundWindow
src1 or *src2* is unbound.

- ```
  template<class P>
  ccPelBuffer<P> cfPelMult(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2, c_Int32 output_mode);
  ```

Applies the user-specified method of handling overflow as it multiplies the pixels in the greatest common region of *src1* and *src2*, writes the resulting pixel values to a new root image, and returns a window that is bound to that image.

Parameters

P Template parameter specifying the type of the windows whose pixels are being multiplied and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

src1 A bound window.

src2 A bound window.

output_mode The method to use to handle overflow. *output_mode* must be one of the following values:

ccPelFunc::eMultDefault (a * b)
ccPelFunc::eMultShft (a * b) >> 8

See **mult_mode** on page 2384.

ccPel::UnboundWindow
src1 or *src2* is unbound.

ccPelFunc::BadParams
output_mode is not a valid multiplication mode.

- ```
  template<class S1, class S2, class D>
  void cfPelMult(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
  ```

Multiplies the pixels in the region of *src1* and *src2* defined by greatest common region of *src1* and *src2* and *dest* using the default output mode (*ccPelFunc::eMultDefault*), and writes the resulting pixel values to *dest*.

Parameters

S1, S2, D — Template parameters that specify the type of the two source windows and the destination window. Not all combinations of *S1*, *S2*, and *D* are instantiated. See *pelfunc.h* to see which instantiations are available.

src1 — A bound window.

src2 — A bound window.

dest — A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, and *dest* will be multiplied and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1* and *src2*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then multiplied and the resulting values are written to *dest*.

Throws

ccPel::UnboundWindow — *src1* or *src2* is unbound.

ccPelFunc::Overlap — The source window overlaps the destination window, but they are not identical (after adjusting for greatest common region).

- ```
  template<class S1, class S2, class D>
  void cfPelMult(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2,
    ccPelBuffer<D>& dest, c_Int32 output_mode);
  ```

  Applies the user-specified method of handling overflow as it multiplies the pixels in the region of *src1* and *src2* defined by greatest common region of *src1* and *src2* and *dest*, and writes the resulting pixels values to *dest*.

  Parameters

  S1, S2, D — Template parameters that specify the type of the two source windows and the destination window. Not all combinations of *S1*, *S2*, and *D* are instantiated. See *pelfunc.h* to see which instantiations are available.

  src1 — A bound window.

  src2 — A bound window.

  dest — A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, and *dest* will be multiplied and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of
  ```

src1 and *src2*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then multiplied and the resulting values are written to *dest*.

output_mode The method to use to handle overflow. *output_mode* must be one of the following values:

ccPelFunc::eMultDefault
ccPelFunc::eMultShft

Throws

ccPel::UnboundWindow
src1 or *src2* is unbound.

ccPelFunc::Overlap
The source window overlaps the destination window, but they are not identical (after adjusting for greatest common region).

ccPelFunc::BadParams
output_mode is not a valid multiplication mode.

cfPelMultAdd()

```
#include <ch_cvl/pelfunc.h>

cfPelMultAdd();
```

Global function to multiply the pixels of two windows and add the pixels of a third window.

Note When this function computes the greatest common region of three images it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

cfPelMultAdd

```
template<class P>
ccPelBuffer<P> cfPelMultAdd(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2,
  const ccPelBuffer_const<P>& src3);

template<class P>
ccPelBuffer<P> cfPelMultAdd(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2,
  const ccPelBuffer_const<P>& src3,
  c_Int32 output_mode);

template<class S1, class S2, class S3, class D>
void cfPelMultAdd(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2,
  const ccPelbuffer_const<S3>& src3,
  ccPelBuffer<D>& dest);

template<class S1, class S2, class S3, class D>
void cfPelMultAdd(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2,
  const ccPelbuffer_const<S3>& src3,
  ccPelBuffer<D>& dest,
  c_Int32 output_mode);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelMultAdd(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2,
    const ccPelBuffer_const<P>& src3);
  ```

  Multiplies the pixels in the greatest common region of *src1*, *src2* and *src3* using the default output mode (*ccPelFunc::eMultAddDefault*), adds the pixels in *src3*, then writes the resulting pixels values to a new root image, and returns a window that is
  ```

bound to that image.

Parameters

P	Template parameter specifying the type of the windows whose pixels are being multiplied and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
src1	A bound window.
src2	A bound window.
src3	A bound window.

Throws

ccPel::UnboundWindow
src1, *src2,* or *src3* is unbound.

-
```
template<class P>
ccPelBuffer<P> cfPelMultAdd(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2,
  const ccPelBuffer_const<P>& src3,
  c_Int32 output_mode);
```

Applies the user-specified method of handling overflow as it operates on the pixels in the greatest common region of *src1*, *src2* and *src3,* multiplies *src1* and *src2*, adds the pixels in *src3*, writes the resulting pixel values to a new root image, and returns a window that is bound to that image.

Parameters

P	Template parameter specifying the type of the windows whose pixels are being multiplied and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
src1	A bound window.
src2	A bound window.
src3	A bound window.
output_mode	The method to use to handle overflow. *output_mode* must be one of the following values: *ccPelFunc::eMultAddDefault* *ccPelFunc::eMultAddShft* See **mult_add_mode** on page 2385.

ccPel::UnboundWindow
src1, *src2,* or *src3* is unbound.

ccPelFunc::BadParams
output_mode is not a valid mode.

- ```
  template<class S1, class S2, class S3, class D>
  void cfPelMultAdd(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2,
    const ccPelbuffer_const<S3>& src3,
    ccPelBuffer<D>& dest);
  ```

Multiplies the pixels in the region of *src1* and *src2* and adds the pixels in *src3*, defined by greatest common region of *src1*, *src2*, *src3* and *dest* using the default output mode (*ccPelFunc::eMultAddDefault*), and writes the resulting pixel values to *dest*.

Parameters

S1, S2, S3, D — Template parameters that specify the type of the three source windows and the destination window. Not all combinations of *S1*, *S2*, *S3*, and *D* are instantiated. See *pelfunc.h* to see which instantiations are available.

src1 — A bound window.

src2 — A bound window.

src3 — A bound window.

dest — A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, *src3* and *dest* will be used and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1, src2* and *src3*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then multiplied, *src3* is added, and the resulting values are written to *dest*.

Throws

ccPel::UnboundWindow
src1, *src2,* or *src3* is unbound.

ccPelFunc::Overlap
The source window overlaps the destination window, but they are not identical (after adjusting for greatest common region).

- ```
  template<class S1, class S2, class S3, class D>
  void cfPelMultAdd(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2,
  ```

```
const ccPelbuffer_const<S3>& src3,
ccPelBuffer<D>& dest,
c_Int32 output_mode);
```

Applies the user-specified method of handling overflow as it multiplies the pixels in the region of *src1* and *src2* and adds the pixels in *src3*, defined by greatest common region of *src1*, *src2*, *src3* and *dest,* and writes the resulting pixels values to *dest*.

Parameters

S1, S2, S3, D — Template parameters that specify the type of the three source windows and the destination window. Not all combinations of *S1*, *S2*, *S3*, and *D* are instantiated. See *pelfunc.h* to see which instantiations are available.

src1 — A bound window.

src2 — A bound window.

src3 — A bound window.

dest — A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, *src3* and *dest* will be used and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1, src2* and *src3*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then multiplied, *src3* is added, and the resulting values are written to *dest*.

output_mode — The method to use to handle overflow. *output_mode* must be one of the following values:

ccPelFunc::eMultAddDefault (a * b +c)
ccPelFunc::eMultAddShft ((a * b) >> 8) + c

See **mult_add_mode** on page 2385.

Throws

ccPel::UnboundWindow
src1, *src2,* or *src3* is unbound.

ccPelFunc::Overlap
The source window overlaps the destination window, but they are not identical (after adjusting for greatest common region).

ccPelFunc::BadParams
output_mode is not a valid mode.

cfPelNoShare()

```
#include <ch_cvl/pelfunc.h>

cfPelNoShare();
```

Global function that provides a root image to which only one window is bound.

cfPelNoShare

```
template<class P> void cfPelNoShare(ccPelBuffer<P>& win);
```

If *win* is unbound or if the reference count of its root image is already 1 (the desired effect of this function), there is no effect. Otherwise, the root image is copied and *win* is rebound to the copied root image. The original root offset and size of *win* are preserved.

Parameters

P	Template parameter specifying the type of the pixels in *win*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
win	A window that the function will modify, if necessary, so that it is the only window bound to its root image.

cfPelNoShare()

cfPelPrint()

```
#include <ch_cvl/pelfunc.h>

cfPelPrint();
```

Global function to print the contents of a window to an output stream.

cfPelPrint

```
template<class P> void cfPelPrint(
  const ccPelBuffer_const<P>& win, ccCvlOStream& stream);
```

Prints the pixel values of a window to the specified stream buffer. Pixel values are printed with white space separation; each row of the window is printed on its own line.

Parameters

P	Template parameter specifying the type of the pixels in *win*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.
win	A bound window.
stream	A stream buffer.

Notes

You are expected to set up *stream* with the formatting options you want (radix, width, precision, and so on).

Throws

ccPel::UnboundWindow
win is unbound.

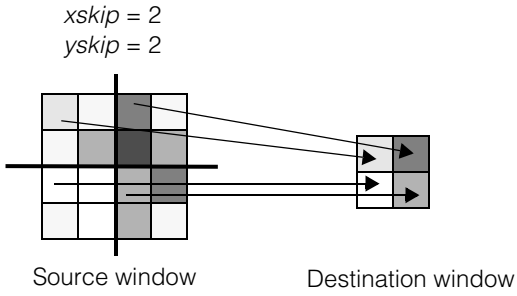
cfPelPrint()

cfPelSample()

```
#include <ch_cvl/pelfunc.h>

cfPelSample();
```

Global function to subsample the pixels in a window to produce a window of lower resolution and reduced size. The following figure shows an example of subsampling.



cfPelSample() subsamples a window by dividing its pixels into blocks and selecting the pixel closest to each block's center. When there is no pixel at the exact center of a subsampling block (as is the case in the figure), the pixel to the upper left of center is selected.

cfPelSample

```
template<class P>
ccPelBuffer<P> cfPelSample(const ccPelBuffer_const<P>&,
  c_Int32 xskip, c_Int32 yskip);

template<class S, class D>
void cfPelSample(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest, c_Int32 xskip, c_Int32 yskip);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelSample(const ccPelBuffer_const<P>&,
    c_Int32 xskip, c_Int32 yskip);
  ```

  Subsamples *src*, writes the selected pixel value from each subsampling block to a new root image, and returns a window that is bound to that image.
  ```

Parameters

P — Template parameter specifying the type of the window whose pixels are being subsampled and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, **c_UInt32**, **ccPackedRGB16Pel**, or **ccPackedRGB32Pel**.

src — A bound window.

xskip, yskip — The x and y dimensions of the subsampling blocks. *xskip* and *yskip* must be greater than 0.

Throws

ccPel::UnboundWindow
src is unbound.

- ```
  template<class S, class D>
  void cfPelSample(const ccPelBuffer_const<S>& src,
    ccPelBuffer<D>& dest, c_Int32 xskip, c_Int32 yskip);
  ```

  Subsamples *src* and writes the selected pixel values to *dest*.

  Parameters

  S, *D* — Template parameters specifying the type of the source and destination windows. *S* and *D* are always the same, and must be one of **c_UInt8**, **c_UInt16**, **c_UInt32**, **ccPackedRGB16Pel**, or **ccPackedRGB32Pel**.

  src — A bound window.

  dest — A bound or unbound window.

  If *dest* is bound, the function calculates the size that *src* would be after sampling, then determines the greatest common region of this reduced *src* with *dest* to determine how much of *src* is to be sampled.

  If *dest* is unbound, a new root image is allocated to accommodate the results of the subsampling, and *dest* is bound to that root image. All the pixels in *src* are subsampled and the results are written to *dest*.

  When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

  xskip, yskip — The x and y dimensions of the subsampling blocks. *xskip* and *yskip* must be greater than 0.
  ```

cfPelSet()

```
#include <ch_cvl/pelfunc.h>

cfPelSet();
```

Global function to set each pixel in a window to a specified value.

cfPelSet

```
template<class P> void cfPelSet(
  const ccPelBuffer_const<P>& win, P val);
```

Sets the pixels in a window to a specified value.

Parameters

P	Template parameter specifying the type of the pixels in *win* and of the value to which those pixels will be set. *P* must be one of **c_UInt8**, **c_UInt16**, **c_UInt32**, **double**, **ccPackedRGB16Pel**, **ccPackedRGB32Pel**.
win	A bound window.
val	The value to which each pixel will be set.

Throws

ccPel::UnboundWindow
win is unbound.

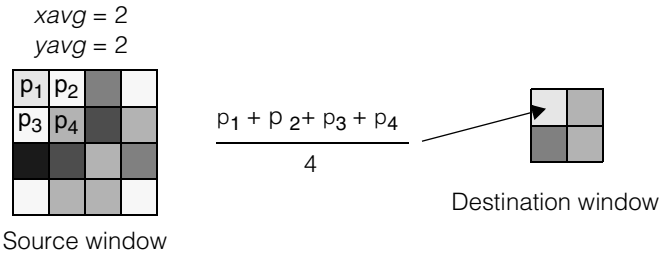
cfPelSet()

cfPelSpatialAvg()

```
#include <ch_cvl/pelfunc.h>

cfPelSpatialAvg();
```

Global function to perform a spatial averaging of the pixels in a window to produce a window of lower resolution and reduced size. The following figure shows an example of subsampling.



Spatial averaging divides the source window into blocks of the size indicated by the *xavg* and *yavg* arguments. The pixel values from each pixel in the block are summed, and the result is divided by the number of pixels in the block. The averaged pixel value from the block of pixels is then applied to a single pixel in the destination window. Partial source blocks are ignored during spatial averaging.

cfPelSpatialAvg

```
template<class P, class I>
ccPelBuffer<P> cfPelSpatialAvg(
  const ccPelBuffer_const<P>&, c_Int32 xavg, c_Int32 yavg,
  I dummy);

template<class S, class D, class I>
void cfPelSpatialAvg(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest, c_Int32 xavg, c_Int32 yavg,
  I dummy);
```

- ```
  template<class P, class I>
  ccPelBuffer<P> cfPelSpatialAvg(
    const ccPelBuffer_const<P>&, c_Int32 xavg, c_Int32 yavg,
    I dummy);
  ```

  Performs a spatial averaging of the pixels in *src* using sampling blocks of size *xavg* by *yavg*, writes the average pixel value of each block to a new root image, and returns a window that is bound to that image. The value of *dummy* is not used; only the type specified by the template parameter *I* is required by the function.
  ```

Parameters

P — Template parameter specifying the type of the window whose pixels are being averaged and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

I — This template parameter specifies a data type that the function uses internally. *I* must be **c_UInt32**.

src — A bound window.

xavg, *yavg* — The x and y dimensions of the sampling blocks. *xavg* and *yavg* must be greater than 0.

Throws

ccPel::UnboundWindow
src is unbound.

- ```
  template<class S, class D, class I>
  void cfPelSpatialAvg(const ccPelBuffer_const<S>& src,
    ccPelBuffer<D>& dest, c_Int32 xavg, c_Int32 yavg,
    I dummy);
  ```

Performs a spatial averaging of the pixels in *src* using sampling blocks of size *xavg* by *yavg* and writes the average pixel value of each block to *dest*. The value of *dummy* is not used; only the type specified by the template parameter *I* is required by the function.

Parameters

S, *D* — Template parameters specifying the type of the source and destination windows. *S* and *D* are always the same, and must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

I — This template parameter specifies a data type that the function uses internally. *I* must be **c_UInt32**.

src — A bound window.

dest — A bound or unbound window.

If *dest* is unbound, a new root image is allocated to accommodate the results of the spatial averaging, and *dest* is bound to the root image. All the pixels in *src* are then averaged and the results are written to *dest*.

If *dest* is bound, the function calculates the size that *src* would be after spatial averaging, then determines the greatest common region of this reduced *src* with *dest* to determine how much of *src* is to be averaged.
  ```

When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

xavg, yavg — The x and y dimensions of the sampling blocks. *xavg* and *yavg* must be greater than 0.

cfPelSpatialAvg()

cfPelSub()

```
#include <ch_cvl/pelfunc.h>

cfPelSub();
```

Global function to subtract the pixels of two windows.

Note When this function computes the greatest common region of two images it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

cfPelSub

```
template<class P>
ccPelBuffer<P> cfPelSub(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);

template<class P>
ccPelBuffer<P> cfPelSub(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2, c_Int32 output_mode);

template<class S1, class S2, class D>
void cfPelSub(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);

template<class S1, class S2, class D>
void cfPelSub(const ccPelBuffer_const<S1>& src1,
  const ccPelBuffer_const<S2>& src2,
  ccPelBuffer<D>& dest, c_Int32 output_mode);
```

-
```
template<class P>
ccPelBuffer<P> cfPelSub(
  const ccPelBuffer_const<P>& src1,
  const ccPelBuffer_const<P>& src2);
```

In the greatest common region of *src1* and *src2*, subtracts the pixels in *src2* from the pixels in *src1* using the default output mode (*ccPelFunc:: eSubDefault)*, writes the resulting pixel values to a new root image, and returns a window that is bound to that image.

Parameters

P Template parameter specifying the type of the windows whose pixels are being subtracted and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

src1 A bound window.

src2 A bound window.

Throws

ccPel::UnboundWindow
src1 or *src2* is unbound.

- ```
  template<class P>
  ccPelBuffer<P> cfPelSub(
    const ccPelBuffer_const<P>& src1,
    const ccPelBuffer_const<P>& src2, c_Int32 output_mode);
  ```

  In the greatest common region of *src1* and *src2*, applies the user-specified method of handling overflow as it subtracts the pixels in *src2* from the pixels in *src1*, writes the resulting pixel values to a new root image, and returns a window that is bound to that image.

  Parameters

  P Template parameter specifying the type of the windows whose pixels are being added and of the window that the function returns. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  src1 A bound window.

  src2 A bound window.

  output_mode The method to use to handle overflow. *output_mode* must be one of the following values:

  ccPelFunc::eSubDefault
  ccPelFunc::eSubAbsDefault
  ccPelFunc::eSubAbs
  ccPelFunc::eSubZero
  ccPelFunc::eSubShft
  ccPelFunc::eSubNabs
  ccPelFunc::eSubLimit

  See **sub_mode** on page 2383.

  Throws

  ccPel::UnboundWindow
  src1 or *src2* is unbound.
  ```

ccPelFunc::BadParams
output_mode is not a valid subtraction mode.

- ```
  template<class S1, class S2, class D>
  void cfPelSub(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2, ccPelBuffer<D>& dest);
  ```

  In the greatest common region of *src1*, *src2*, and *dest*, subtracts the pixels in *src2* from the pixels in *src1* using the default output mode (*ccPelFunc:: eSubDefault)* and writes the resulting pixel values to *dest*.

  Parameters

  S1, S2, D — Template parameters that specify the type of the two source windows and the destination window. *S1, S2,* and *D* are always the same type, and must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  src1 — A bound window.

  src2 — A bound window.

  dest — A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, and *dest* will be subtracted and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1* and *src2*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then subtracted and the resulting values are written to *dest*.

  Throws

  ccPel::UnboundWindow
  src1 or *src2* is unbound.

- ```
  template<class S1, class S2, class D>
  void cfPelSub(const ccPelBuffer_const<S1>& src1,
    const ccPelBuffer_const<S2>& src2,
    ccPelBuffer<D>& dest, c_Int32 output_mode);
  ```

  In the greatest common region of *src1*, *src2*, and *dest*, applies the user-specified method of handling overflow as it subtracts the pixels in *src2* from the pixels in *src1* and writes the resulting pixel values to *dest*.

Parameters

S1, S2, D
Template parameters that specify the type of the two source windows and the destination window. *S1, S2,* and *D* are always the same type, and must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

src1
A bound window.

src2
A bound window.

dest
A bound or unbound window. If *dest* is bound, the pixels in the greatest common region of *src1*, *src2*, and *dest* will be subtracted and written to *dest*. If *dest* is unbound, a new root image is allocated whose size is equal to the greatest common region of *src1* and *src2*, after which *dest* is bound to the new root image. The pixels in the greatest common region of *src1* and *src2* are then subtracted and the resulting values are written to *dest*.

output_mode
The method to use to handle overflow. *output_mode* must be one of the following values:

ccPelFunc:: eSubDefault
ccPelFunc::eSubAbsDefault
ccPelFunc::eSubAbs
ccPelFunc::eSubZero
ccPelFunc::eSubShft
ccPelFunc::eSubNabs
ccPelFunc::eSubLimit

See **sub_mode** on page 2383.

Throws

ccPel::UnboundWindow
src1 or *src2* is unbound.

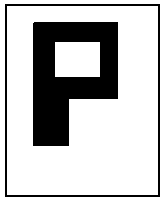
ccPelFunc::BadParams
output_mode is not a valid subtraction mode.

cfPelTranspose()

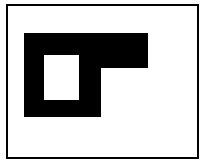
```
#include <ch_cvl/pelfunc.h>

cfPelTranspose();
```

Global function to transpose the pixels in a window. The following figure shows an example of a transposed window.



Source window



Transposed window

cfPelTranspose

```
template<class P>
ccPelBuffer<P> cfPelTranspose(
  const ccPelBuffer_const<P>& src);

template<class S, class D>
void cfPelTranspose(const ccPelBuffer_const<S>& src,
  ccPelBuffer<D>& dest);
```

- ```
  template<class P>
  ccPelBuffer<P> cfPelTranspose(
    const ccPelBuffer_const<P>& src);
  ```

  Transposes the pixels in *src*, copies them to a new root image, and returns a window that is bound to that image.

  Parameters

  P — Template parameter specifying the type of the window to be returned by the function and of the pixels in *src*. *P* must be one of **c_UInt8**, **c_UInt16**, or **c_UInt32**.

  src — A bound window.

  Throws

  ccPel::UnboundWindow — *src* is unbound.
  ```

- ```
  template<class S, class D>
  void cfPelTranspose(const ccPelBuffer_const<S>& src,
    ccPelBuffer<D>& dest);
  ```

  Transposes the pixels in *src* and writes them to *dest*. If the same window is specified for *src* and *dest*, the operation is performed in place. For this to work, the window must be square.

  Parameters

  S — Template parameter specifying the type of the pixels in *src*. *S* must be one of **c_UInt8**, **c_UInt16** or **c_UInt32**.

  D — Template parameter specifying the type of the pixels in *dest*. *D* must be the same type as *S*.

  src — A bound window.

  dest — A bound or unbound window. If *dest* is bound, the pixels in the region of *src* determined by the greatest common region of *src* and *dest* are transposed and written to *dest*. If *dest* is unbound, a root image for *dest* is allocated and all the pixels in *src* are transposed and copied to *dest*.

  When this function computes the greatest common region of *src* and *dest* it considers each image's image coordinate offset. For more information on image coordinates and image coordinate offset, see the chapter *Images and Coordinates* in the *CVL User's Guide*.

  Throws

  ccPel::UnboundWindow — *src* is unbound.

  ccPelFunc::Overlap — *src* and *dest* overlap but are not identical, or are identical but not square.
  ```

cfPMInspectDisplayFeatures()

```
#include <ch_cvl/pmifproc.h>

cfPMInspectDisplayFeatures();
```

Global function to create a graphical display of features detected by PatInspect.

cfPMInspectDisplayFeatures

```
void cfPMInspectDisplayFeatures(ccUITablet& tablet,
  ccPMInspectSimpleBoundaryDiffData& data,
  bool showMatch = true, bool showExtra = true,
  bool showMissing = true, bool showConnect = true,
  ccColor cmatch = ccColor::blueColor(),
  ccColor cextra = ccColor::redColor(),
  ccColor cmissing = ccColor::yellowColor(),
  ccColor cconnect = ccColor::whiteColor(),
  ccColor cpattern = ccColor::greenColor(),
  cc2Xform pose=cc2Xform::I);
```

Parameters

tablet	A **ccUITablet** into which the graphics are drawn.
data	The feature difference data returned by **ccPMInspectResult::getSimpleBoundaryDiff()**.
showMatch	If true, show the matched features' boundary points.
showExtra	If true, show the extra features' boundary points.
showMissing	If true, show the missing features' boundary points.
showConnect	If true, connect boundary points.
cmatch	The color in which to draw the matched features's boundary points.
cextra	The color in which to draw the extra features's boundary points.
cmissing	The color in which to draw the missing features's boundary points.
cconnect	The color in which to draw the line segments connecting boundary points.
cpattern	The color in which to draw the trained pattern's feature boundary points that correspond to match boundary points in the image. If *showMatch* is false, the pattern's feature boundary points are not displayed.

pose	A **cc2Xform** that tells how to transform the display of the feature boundary points. You can specify the pose returned by **ccPMInspectResult::pose()** to display the feature boundary points for a particular pattern instance in a run-time image.

cfPolarTransformImage()

```
#include <ch_cvl/polar.h>

cfPolarTransformImage();
```

Global function to perform a polar transformation on an input image. Polar transformation consists of sampling the source image at the grid location specified by the sampling parameters, and placing the samples into the corresponding pixels of the destination image.

If the destination image is unbound, a new **ccPelBuffer** of the appropriate size is allocated and bound to the specified destination with an offset of (0, 0).

If the destination image is bound, it must not share any of the source pixels that are sampled. Samples are written to the destination as if the following steps had been executed:

1. A temporary destination image, *TMP*, is created using the rules for unbound destination images.
2. The function **cfPelCopy(TMP, destination)** is used to transfer pixels from *TMP* to the bound destination. This operation only copies pixels within the Greatest Common Rectangle (GCR) of *TMP* and *destination*.

In some cases, it may be necessary to shift the image offset of the destination image to achieve the desired result. The **cfPolarTransformImage()** function does not resize, reallocate, or alter the offset of a destination.

See also the *Polar Coordinate Transformation Tools* section of the *Image Transformation Tools* chapter in the *CVL Vision Tools Guide*.

cfPolarTransformImage

```
template<class T> void
cfPolarTransformImage(
  const ccPelBuffer_const<T>& srcImage,
  ccPelBuffer<T>& dstImage,
  const ccPolarSamplingParams& params);

template<class T> ccPelBuffer<T>
cfPolarTransformImage(
  const ccPelBuffer_const<T>& srcImage,
  const ccPolarSamplingParams& params);
```

Notes

CVL provides instantiations of both overloads for 8-bit source and destination images.

- ```
  template<class T> void
  cfPolarTransformImage(
    const ccPelBuffer_const<T>& srcImage,
    ccPelBuffer<T>& dstImage,
    const ccPolarSamplingParams& params);
  ```

  Performs a polar transformation of an image using the supplied sampling parameters.

  Parameters

  srcImage Source image on which to perform the polar transformation.

  dstImage Destination image in which to store the results of the polar transformation.

  params Sampling parameters that specify the grid location of the source image. The sampling parameters must be specified in the client coordinate system of the source image.

  Notes

  The polar transformation sets the destination image's client coordinate transform to **cc2Xform(cc2Matrix(1, 0, 0, 1), cc2Vect(0, 0))**. Use the **ccPolarSamplingParams** methods **mapImagePosition()** and **mapPolarPosition()** to map between source and polar images.

  Throws

  ccPolarTransDefs::UnboundWindow
  srcImage is an unbound pel buffer.

  ccPolarTransDefs::Clipped
  Clipping has occurred (that is, any of the sampling points require data that is outside of the source image). Use **ccPolarSamplingParams::willClip()** to test whether clipping will occur (without calling **cfPolarTransformImage()**).

- ```
  template<class T> ccPelBuffer<T>
  cfPolarTransformImage(
    const ccPelBuffer_const<T>& srcImage,
    const ccPolarSamplingParams& params);
  ```

  Performs a polar transformation of an image using the supplied sampling parameters and returns a new destination image. Does not require a supplied destination image.

  Parameters

  srcImage The source image on which to perform the polar transformation.

  params Sampling parameters that specify the grid location of the source image. The sampling parameters must be specified in the client coordinate system of the source image.

cfPolylineShapeModel()

```
#include <ch_cvl/shapemod.h>

cfPolylineShapeModel();
```

Global function to get a polyline shape model corresponding to a specified chain in a featurelet chain set.

cfPolylineShapeModel

```
ccPolylineShapeModelPtrh cfPolylineShapeModel(
  const ccFeatureletChainSet &featureletChainSet,
  c_Int32 chainIndex);
```

Retrieves the polyline shape model corresponding to the specified chain in the supplied featurelet chain set. The returned polyline simply connects the positions of the featurelets in order along the chain, and therefore has the same handedness and open/closed state as the featurelet chain.

Featurelet angles are used only indirectly to determine the *polarity* of the returned model. Specifically, the polarity is completely dependent on the value returned by **proportionPositiveCrossProduct()** for the specified chain. The polarity of the returned shape model is reversed if and only if this value is less than 0.5. The polarity of the returned shape model is ignored if and only if this value is exactly equal to 0.5.

The *weight* and *magnitude* of the returned shape model are the average weight and average magnitude of the featurelets in the chain, respectively.

Throws

ccFeatureletDefs::BadParams
chainIndex is less than zero or
greater than or equal to **featureletChainSet.numChains()**

cfPolylineShapeModel()

cfPolySetNearestPoints()

```
#include <ch_cvl/shapgeom.h>

cfPolySetNearestPoints();
```

Global function to return the nearest two points between two **ccPolyline** object sets. The returned points *p1* and *p2*, are on the **ccPolyline** sets *shapeTree1* and *shapeTree2*, respectively. The supplied **ccGeneralShapeTree** objects are assumed to have only **ccPolyline** objects as children.

cfPolySetNearestPoints

```
void cfPolySetNearestPoints(
  const ccGeneralShapeTree &shapeTree1,
  const ccGeneralShapeTree &shapeTree2,
  cc2Vect &p1, cc2Vect &p2);
```

Parameters

shapeTree1	The first **ccPolyline** object set.
shapeTree2	The second **ccPolyline** object set.
p1	The returned nearest point on *shapeTree1*.
p2	The returned nearest point on *shapeTree2*.

Throws

ccShapesError::BadParams
: Either of the two supplied shape trees includes objects of any type other than **ccPolyline**.

ccShapesError::EmptyShape
: Either of the two supplied shape trees is empty.

Notes

If more than one pair of points have the same nearest distance, this function returns an arbitrary pair among them.

cfPolySetNearestPoints()

cfProjectImage()

```
#include <ch_cvl/project.h>

cfProjectImage();
```

Global function to construct a 1-dimensional projection image using a 2-dimensional input image and a **ccAffineSamplingParams**.

cfProjectImage

```
template<class S>
void cfProjectImage(
  const ccPelBuffer_const<S>& srcImage,
  ccPelBuffer<c_UInt32>& dstImage,
  const ccAffineSamplingParams& params,
  c_UInt32& nPelsPerBin);

template<class S>
ccPelBuffer<c_UInt32> cfProjectImage(
  const ccPelBuffer_const<S>& srcImage,
  const ccAffineSamplingParams& params,
  c_UInt32& nPelsPerBin);

template<class S>
void cfProjectImage(
  const ccPelBuffer_const<S>& srcImage,
  ccPelBuffer<c_UInt32>& dstImage,
  ccPelBuffer<c_UInt32>& weights,
  const ccAffineSamplingParams& params,
  c_UInt32 &nPelsPerBin,
  bool& clipped);
```

-
```
template<class S>
void cfProjectImage(
  const ccPelBuffer_const<S>& srcImage,
  ccPelBuffer<c_UInt32>& dstImage,
  const ccAffineSamplingParams& params,
  c_UInt32& nPelsPerBin);
```

Projects the supplied input image into the supplied destination image using the supplied **ccAffineSamplingParams**.

Parameters

srcImage	The source image. *srcImage* must be of type **c_UInt8**.
dstImage	The destination image. If *dstImage* is unbound, the function will allocate a root image and bind *dstImage* to it.
params	A **ccAffineSamplingParams** that defines the projection region.

nPelsPerBin A reference to a **c_UInt32** into which the function will place the number of pixels in *srcImage* which were summed to form each pixel in *dstImage*.

Throws

ccAffTransDefs::Clipped
The projection region defined by *params* is clipped by *srcImage*. See below for a version of this function that allows clipping.

ccAffProjImgDefs::UnboundWindow
The source image is unbound.

ccAffineSamplingParams::NotImplemented
params.interpolation is *ccAffineSamplingParams::eBilinearApprox* or *ccAffineSamplingParams::eHighPrecision* and *srcImage.rowUpdate()* or *srcImage.height()* is greater than or equal to 32768.

Notes

The **cc2XForm** for *dstImage* will map locations in the *dstImage* client coordinate system to the corresponding locations within the affine rectangle specified by *params* in the client coordinates of *srcImage*.

- ```
  template<class S>
  ccPelBuffer<c_UInt32> cfProjectImage(
    const ccPelBuffer_const<S>& srcImage,
    const ccAffineSamplingParams& params,
    c_UInt32& nPelsPerBin);
  ```

Returns a newly constructed and bound **ccPelBuffer<c_UInt32>** into which the supplied source image has been projected according to the supplied **ccAffineSamplingParams**.

Parameters

srcImage The source image. *srcImage* must be of type **c_UInt8**.

params A **ccAffineSamplingParams** that defines the projection region.

nPelsPerBin A reference to a **c_UInt32** into which the function will place the number of pixels in *srcImage* which were summed to form each pixel in *dstImage*.

Throws

ccAffTransDefs::Clipped
The projection region defined by *params* is clipped by *srcImage*. See below for a version of this function that allows clipping.
  ```

ccAffProjImgDefs::UnboundWindow
The source image is unbound.

ccAffineSamplingParams::NotImplemented
params.interpolation is *ccAffineSamplingParams::eBilinearApprox* or *ccAffineSamplingParams::eHighPrecision* and *srcImage.rowUpdate()* or *srcImage.height()* is greater than or equal to 32768.

- ```
  template<class S>
  void cfProjectImage(
    const ccPelBuffer_const<S>& srcImage,
    ccPelBuffer<c_UInt32>& dstImage,
    ccPelBuffer<c_UInt32>& weights,
    const ccAffineSamplingParams& params,
    c_UInt32 &nPelsPerBin,
    bool& clipped);
  ```

  Projects the supplied input image into the supplied destination image using the supplied **ccAffineSamplingParams**.

  If the affine rectangle defined by **ccAffineSamplingParams** clips the source image, this function computes the projection based on the clipped source window, and *clipped* is set to true to indicated that clipping occurred. If clipping took place, the returned weights image contains the number of pixels that were summed to compute the corresponding pixel in the projection image.

  Parameters

  srcImage The source image. *srcImage* must be of type **c_UInt8**.

  dstImage The destination image. If *dstImage* is unbound, the function will allocate a root image and bind *dstImage* to it.

  weights If clipping took place, *weights* is filled with values that indicate how many sample pels were summed to compute the corresponding projection pel. If clipping did not take place, this buffer is not changed.

  params A **ccAffineSamplingParams** that defines the projection region.

  nPelsPerBin A reference to a **c_UInt32** into which the function will place the number of pixels in *srcImage* which were summed to form each pixel in *dstImage*.

  clipped True if clipping occurred, false otherwise.

  Throws
  ```

ccAffProjImgDefs::UnboundWindow
: The source image is unbound.

ccPel::BadCoord
: The affine rectangle is completely off the source window (that is, the intersection of the affine rectangle and the source window is empty).

ccAffineSamplingParams::NotImplemented
: *params.interpolation* is *ccAffineSamplingParams::eBilinearApprox* or *ccAffineSamplingParams::eHighPrecision* and *srcImage.rowUpdate()* or *srcImage.height()* is greater than or equal to 32768.

cfRasterize()

```
#include <ch_cvl/raster.h>

cfRasterize();
```

Global function to rasterize an image of a sampled **ccRegionTree** into a pel buffer. The **ccRegionTree** is first clipped by the pel buffer window and then sampled in image coordinates by the **ccShape::sample()** method using the specified sampling parameters.

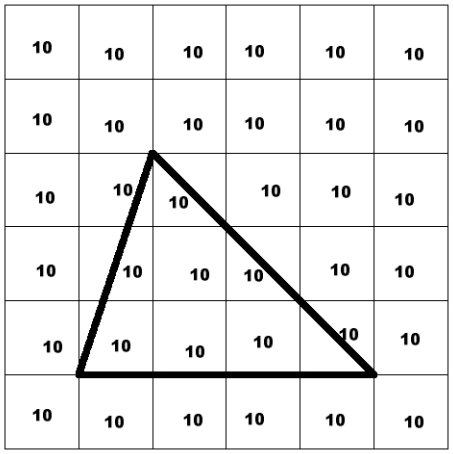
A *raster* is a scan pattern (as of the electron beam in a cathode-ray tube) in which an area is scanned from side to side in lines from top to bottom. To *rasterize* means to scan the image rows (or columns) and calculate the grey level of each pixel individually.

The main differences between rasterizing and drawing are the following:

- A rasterized shape is filled in. That is, more than just its boundary is shown.
- A rasterized shape is converted into a form that may be displayed on a grid of pixels. This can produce digitization errors near the edges of the shape because pixels have finite size.

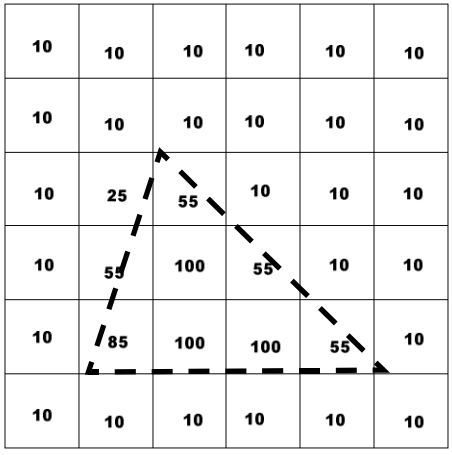
The **cfRasterize()** function performs this process by logically scanning each image row and determining whether each pixel should be assigned a new value on a per-pixel basis. The scan is performed *logically*, rather than physically, because the function takes some shortcuts, such as ignoring complete rows or finding the starting and ending columns for each row. The pixels of solid regions are set to the foreground grey level *fg*. Pixels of hole regions are not changed. Phantom holes (that is, hole regions that are not children of some enclosing solid region) are not rasterized.

The *boundaryFillMode* parameter specifies how to handle pel values that are on the boundary between solid and hole regions. The figures on the next few pages illustrate the various boundary fill mode options.

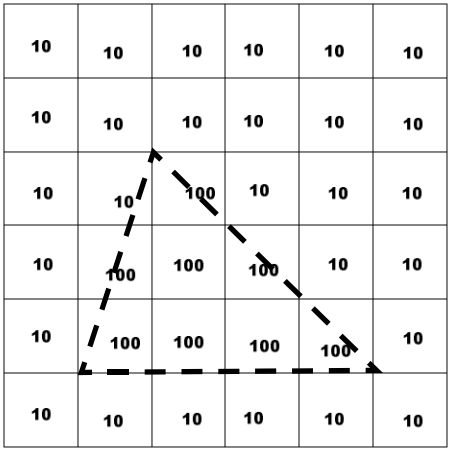


This figure shows a 6x6 pel buffer whose pels all contain a value of 10. A polygon is superimposed on this image to show the **ccRegionTree** boundary to be rasterized on this pel buffer.

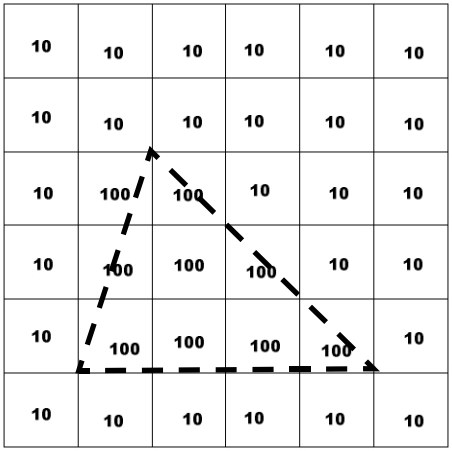
The *boundaryFillMode* is specified as the fourth parameter passed to **cfRasterize()**.



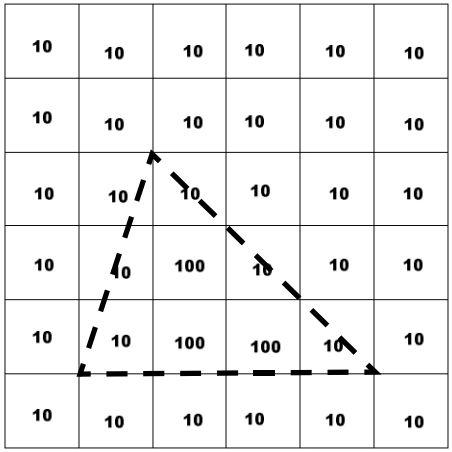
This figure shows the result of using an interpolated value for pels that are on the boundary between solid and hole regions. The interpolated value is a grey level that interpolates the existing grey level and the foreground level using the fraction of the pixel area that the region tree covers. This is the anti-aliasing mode; it is also the default.You would achieve this effect by calling **cfRasterize(region, buffer, 100, ccRasterizationDefs::eUseInterpolatedPelValue)**.



This figure shows the result of using the value covering the largest portion of the pixel for pels that are on the boundary between solid and hole regions. This setting uses the foreground level if at least half of the boundary pixel is covered by the region tree to be rasterized. Otherwise, the value is not changed. You would achieve this effect by calling **cfRasterize(region, buffer, 100, ccRasterizationDefs::eUseLargestPortionPelValue)** .



This figure shows the result of using the foreground value for pels that are on the boundary between solid and hole regions. You would achieve this effect by calling **cfRasterize(region, buffer, 100, ccRasterizationDefs::eUseForegroundPelValue)**.



This figure shows the result of using the background value for pels that are on the boundary between solid and hole regions. You would achieve this effect by calling **cfRasterize(region, buffer, 100, ccRasterizationDefs::eUseBackgroundPelValue)**.

cfRasterize

```
void cfRasterize(const ccRegionTree &region,
  const ccPelBuffer<c_UInt8> &pelbuf,
  c_UInt8 fg,
  ccRasterizationDefs::BoundaryFillMode boundaryFillMode =
    ccRasterizationDefs::kDefaultBoundaryFillMode,
  const ccShape::ccSampleParams & samplingParams =
    ccRasterizationDefs::RasterizeSampParams());
```

Parameters

region The region tree to draw, specified in client coordinates.

pelbuf The pel buffer into which to draw the region tree.

fg Foreground grey level. This value is assigned to pixels of solid regions.

boundaryFillMode

Specifies how to handle pixel values that are on the boundary between solid and hole regions. Possible values:

Value	Meaning
ccRasterizationDefs:: eUseInterpolatedPelValue	Use a grey level that interpolates the existing grey level and *fg* using the fraction of the pixel area that the region tree covers. This is the anti-aliasing mode. This mode is also the default.
ccRasterizationDefs:: eUseLargestPortionPelValue	Use *fg* if at least half of the boundary pixel is covered by *region*. Otherwise, don't change the pel value.
ccRasterizationDefs:: eUseForegroundPelValue	Use *fg* for boundary pixels.
ccRasterizationDefs:: eUseBackgroundPelValue	Do not change boundary pixel values. Existing pixel values remain unchanged.

samplingParams Parameters used to sample the shape.

Throws

ccPel::UnboundWindow
pelbuf is unbound.

ccShapesError::SampleOverflow
Possible throw from **ccShape::sample()**. See **ccShape::sample()** for details.

Notes

This function requires the boundary and children of the region to be non-self-intersecting. Otherwise, its behavior is not defined.

This function respects the client transform of the supplied pel buffer. Always specify the region in client coordinates.

cfRasterizeContour()

```
#include <ch_cvl/raster.h>

cfRasterizeContour();
```

Global function to rasterize the sampled contour of the specified shape on the specified pel buffer using the specified grey level and thickness values. The shape is sampled in image coordinates by the **ccShape::sample()** method using the specified sampling parameters.

To *rasterize* means to scan the image rows (or columns) and calculate the grey level of each pixel individually. The **cfRasterizeContour()** function performs this process by logically scanning each image row and determining whether each pixel should be assigned a new value.

The **cfRasterizeContour()** function creates a circular stencil with a diameter of *thickness* centered around each sampled point on the pel buffer and slides it along the line between this point and the next sampled point. Any pixels that are touching this stencil are assigned the foreground color value *fg*. The remaining pixels are not changed. Any pixels that are outside the pel buffer window and touching the circular stencil are ignored by means of clipping the circular stencil by the pel buffer window.

cfRasterizeContour

```
void cfRasterizeContour(const ccShape &shape,
  const ccPelBuffer<c_UInt8> &pelbuf,
  c_UInt8 fg,
  double thickness,
  const ccShape::ccSampleParams & samplingParams =
    ccRasterizationDefs::RasterizeSampParams());
```

Parameters

shape	The shape to draw, specified in client coordinates.
pelbuf	The pel buffer into which to draw the sampled shape.
fg	Foreground grey level. This value is assigned to any pixel of the pel buffer that is covered by the stencil.
thickness	Diameter of the circular stencil centered around the sampled points and the connecting lines between them, specified in client coordinates.
samplingParams	Parameters used to sample the shape.

Throws

ccPel::UnboundWindow
pelbuf is unbound.

ccShapesError::BadRadius
: *thickness* is less than or equal to 0.0.

ccShapesError::SampleOverflow
: Possible throw from **ccShape::sample()**. See **ccShape::sample()** for details.

Notes

This function respects the client transform of the supplied **ccPelBuffer**. Always specify the shape and thickness in client coordinates.

cfRealEq()

```
#include <ch_cvl/math.h> // for floating point values
#include <ch_cvl/vector.h> // for vectors
#include <ch_cvl/matrix.h> // for matrices
#include <ch_cvl/xform.h> // for transformations
#include <ch_cvl/units.h> // for angles
#include <ch_cvl/xflinear.h> // for linear transformations

cfRealEq();
```

Global function that tests whether two objects, that contain floating point values, are equal within a specified tolerance.

cfRealEq

```
bool cfRealEq(double x, double y, double epsilon = 1.e-15);

bool cfRealEq(const cc2Vect& v1, const cc2Vect &v2,
  double epsilon = 1.e-15);

bool cfRealEq(const cc3Vect& v1, const cc3Vect& v2,
  double epsilon = 1.e-15);

bool cfRealEq(const cc2Matrix& m1, const cc2Matrix& m2,
  double epsilon = 1.e-15);

bool cfRealEq(const cc1Xform& x1, const cc1Xform& x2,
  double epsilon = 1.e-15);

bool cfRealEq(const cc2Xform& x1, const cc2Xform& x2,
  double epsilon = 1.e-15);

bool cfRealEq(const cc2Rigid& x1, const cc2Rigid& x2,
  double epsilon = 1.e-15);

bool cfRealEq (ccRadian x, ccRadian y,
  ccRadian epsilon = ccRadian(1.e-15));

bool cfRealEq(const cc2XformLinear& x1,
  const cc2XformLinear& x2, double epsilon = 1e-15)
```

- `bool cfRealEq(double x, double y, double epsilon = 1.e-15);`

  Returns true if the two supplied values are equal to each other within the specified tolerance *epsilon*. The default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

Parameters

x	The first value to compare.
y	The second value to compare.
epsilon	The tolerance within which the two values must be equal.

- `bool cfRealEq(const cc2Vect& v1, const cc2Vect &v2, double epsilon = 1.e-15);`

Returns true if the corresponding elements of the two vectors are equal to each other within the specified tolerance *epsilon*. The default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

Parameters

v1	The first vector to test.
v2	The second vector to test.
epsilon	The tolerance within which the two vectors must be equal.

- `bool cfRealEq(const cc3Vect& v1, const cc3Vect& v2, double epsilon = 1.e-15);`

Returns true if the corresponding elements of the two vectors are equal to each other within the specified tolerance *epsilon*. The default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

Parameters

v1	The first vector to test.
v2	The second vector to test.
epsilon	The tolerance within which the two vectors must be equal.

- `bool cfRealEq(const cc2Matrix& m1, const cc2Matrix& m2, double epsilon = 1.e-15);`

Returns true if the corresponding elements of the two matrices are equal to each other within the specified tolerance *epsilon*. The default value of epsilon is 10^{-15}. You can override this value when you call the function.

Parameters

m1	The first matrix to test.
m2	The second matrix to test.

epsilon The tolerance within which the two matrices must be equal.

- ```
  bool cfRealEq(const cc1Xform& x1, const cc1Xform& x2,
    double epsilon = 1.e-15);
  ```

  Returns true if the corresponding elements of the two transformation objects (the scale and the offset values) are equal to each other within the specified tolerance *epsilon*. The default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

  Parameters

  x1 The first transformation object to test.

  x2 The second transformation object to test.

  epsilon The tolerance within which the two transformation objects must be equal.

- ```
  bool cfRealEq(const cc2Xform& x1, const cc2Xform& x2,
    double epsilon = 1.e-15);
  ```

  Returns true if the two transformation objects are equal to each other within the specified tolerance *epsilon.* In this case the function evaluates whether the corresponding elements of the matrix component of the two transformation objects and the corresponding elements of the vector component of the two transformation objects are the same within the specified tolerance. Default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

  Parameters

  x1 The first transformation object to test.

  x2 The second transformation object to test.

  epsilon The tolerance within which the two transformation objects must be equal.

- ```
  bool cfRealEq(const cc2Rigid& x1, const cc2Rigid& x2,
    double epsilon = 1.e-15);
  ```

  Returns true if the two transformation objects are equal to each other within the specified tolerance *epsilon.* In this case the function evaluates whether the corresponding elements of the vector component of the two transformation objects and the sine and cosine of the angular component of the two transformation objects are the same within the specified tolerance. The default value of *epsilon* is 10^{-15}. You can override this value when you call the function.
  ```

Parameters

x1	The first transformation object to test.
x2	The second transformation object to test.
epsilon	The tolerance within which the two transformation objects must be equal.

- ```
  bool cfRealEq (ccRadian x, ccRadian y,
    ccRadian epsilon = ccRadian(1.e-15));
  ```

  Returns true if the two supplied radians are equal to each other within the specified tolerance *epsilon*. The default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

  Parameters

x	The first value to compare.
y	The second value to compare.
epsilon	The tolerance within which the two values must be equal.

- ```
  bool cfRealEq(const cc2XformLinear& x1,
    const cc2XformLinear& x2, double epsilon = 1e-15)
  ```

  Returns true if the two transformation objects are equal to each other within the specified tolerance *epsilon.* In this case the function evaluates whether the corresponding elements of the matrix component of the two transformation objects and the corresponding elements of the vector component of the two transformation objects are the same within the specified tolerance. Default value of *epsilon* is 10^{-15}. You can override this value when you call the function.

  Parameters

x1	The first transformation object to test.
x2	The second transformation object to test.
epsilon	The tolerance within which the two transformation objects must be equal.

cfRegionize()

```
#include <ch_cvl/shapgeom.h>

cfRegionize();
```

Global function to convert a shape into a region tree when possible. If the shape is a closed contour, whether a primitive shape or contour tree, this function returns a singleton solid region tree with the contour as the root boundary. If the shape is a general shape tree, this function returns a new general shape tree generated by applying a regionizing algorithm to the shape's children (see *Regionizing General Shape Trees*, below). For all other shapes, including region trees themselves, this function simply returns a clone of the supplied shape.

Regionizing General Shape Trees

This function performs most of its work when the shape is a general shape tree. In this case, the function determines the nesting configuration of the closed contour children of the shape, merging such children into as few region trees as possible, and returning a new general shape tree made up of these region trees.

This function uses the following algorithm to regionize a shape:

1. Recursively regionize all children.
2. Insert regionized children that are not region trees as the final children of the result tree and do not consider them further.
3. The remaining children are all region trees. Among these, *unwrap* any phantom hole trees and replace them with their immediate children so that only solid region trees remain.
4. Determine the nesting structure and merge the region trees appropriately. What remains is a minimal set of region trees that contain exactly the same set of contours as the original set of region tree children, except for the discarded phantom hole contours. All of these region trees will have solid roots, as merging never introduces phantom holes.
5. Check the status of the *groupSeparateRegions* flag:
    - If *groupSeparateRegions* is false, or if there is only a single merged region tree, add the merged region trees directly as the intial children of the general shape tree result.
    - If *groupSeparateRegions* is true and there are multiple merged region trees, add the merged region trees as children to a newly created and appropriately sized phantom hole, and then add this single region tree as a child of the general shape tree result.

cfRegionize()

cfRegionize

```
ccShapePtrh cfRegionize(const ccShape &shape,
  bool groupSeparateRegions);
```

Parameters

shape The shape to be converted to a region tree.

groupSeparateRegions
Whether to group the merged region trees into a single region or to leave them as separate regions.

• If true (and there are multiple merged region trees):	The merged region trees are added as children to a newly created and appropriately sized phantom hole, and this single region tree is then added as a child to the general shape tree result. Phantom hole roots are automatically computed for sets of disjoint solid region trees. These holes will have **ccRect** boundaries that enclose all of the disjoint region trees, possibly with some padding.
• If false (or if there is only a single merged region tree):	The merged region trees are added directly as children to the general shape tree result.

See also *Regionizing General Shape Trees* on page 3831.

Notes

When regionizing general shape trees that contain region trees, the solid/hole status of the region tree boundaries is only used to detect and discard phantom holes. Beyond this, the solid/hole status is ignored as this function always treats the outermost enclosing boundary as a solid boundary when merging trees.

All closed contours within the shape, whether boundaries within region trees or stand-alone contours, must be non-intersecting. The function makes this assumption when determining the nesting structure of a set of contours. If this

assumption is violated, the resulting structure may be inconsistent. The exceptions to this rule are phantom hole boundaries of region trees. As these are discarded, it does not matter if they intersect other contours.

Shape model properties must not have been assigned to the supplied *shape* or any of its descendants. Always invoke this function *before* assigning model properties to a shape.

This function recursively processes an entire shape hierarchy. However, only region shapes that have the same general shape tree parent are candidates for merging into common region trees. Thus, you can use general shape tree nodes used to partition the merge candidates. In some applications, it may be useful to call **flatten()** before calling **cfRegionize()**.

cfRegionize()

cfRGBExtract()

```
#include <ch_cvl/rgbplane.h>

cfRGBExtract();
```

Global function to generate an 8-bit greyscale image from a 32-bit RGB image.

cfRGBExtract

```
void cfRGBExtract(
  const ccPelBuffer_const<ccPackedRGB32Pel> &src32,
  double rw, double gw, double bw,
  ccPelBuffer<c_UInt8>& dest);
```

Generates an 8-bit greyscale image from a 32-bit RGB image using a linear combination of the three component color planes.

If *dest* is unbound, it is allocated to the same size and offset of *src32*. Only the intersection of the *src32* and *dest* images is extracted.

The parameters *rw, gw*, and *bw* are the weights to be applied to each plane, where:

```
pel = rw*R + gw*G + bw*B
```

In general, rw + gw + bw = 1.0. However, this is not enforced and is not a requirement.

Parameters

src32	A 32-bit RGB color pel buffer.
rw	An double less than 32767 describing the weight to be applied to the red color plane when extracting from *src32*.
gw	An double less than 32767 describing the weight to be applied to the green color plane when extracting from *src32*.
bw	An double less than 32767 describing the weight to be applied to the blue color plane when extracting from *src32*.
dest	An 8-bit greyscale pel buffer with the same dimensions as *src32* to hold the extracted and combined image.

Throws

ccPel::BadParams
If any of *rw, gw,* or *bw* is > 32767.

Notes

This function is accelerated when run on a CPU that supports the MMX registers. However, MMX support is not required, the function runs on any CPU supported by CVL.

CPUs that support the MMX registers include the Intel Pentium II, Pentium III, Pentium 4, Xeon, and Celeron processors. Some but not all members of the original Pentium family included MMX support.

cfRGBPack()

```
#include <ch_cvl/rgbplane.h>

cfRGBPack();
```

Global function to generate a packed 32-bit or 16-bit RGB pel buffer from three separate 8-bit red, green, and blue pel buffers.

cfRGBPack

```
void cfRGBPack(
  const ccPelBuffer_const<c_UInt8>& red,
  const ccPelBuffer_const<c_UInt8>& green,
  const ccPelBuffer_const<c_UInt8>& blue,
  const ccPelBuffer<ccPackedRGB32Pel> &result,
  c_UInt8 alpha=0);

void cfRGBPack(
  const ccPelBuffer_const<c_UInt8>& red,
  const ccPelBuffer_const<c_UInt8>& green,
  const ccPelBuffer_const<c_UInt8>& blue,
  const ccPelBuffer<ccPackedRGB16Pel> &result);
```

This function accepts as input red, green, and blue color plane output from an RGB color camera and packs them into a 16-bit or 32-bit pel buffer.

For best performance:

- *result* should be 8-byte aligned
- *result* should be pre-allocated to the same size as the source color planes
- *red*, *green*, and *blue* should be 4-byte aligned.
- The width of all pel buffers should be multiple of 4.

Notes

This function is accelerated when run on a CPU that supports Intel SSE functionality. However, SSE support is not required, the function runs on any CPU supported by CVL.

CPUs that support SSE functionality include the Intel Pentium III, Pentium 4, Xeon, and Celeron processors.

cfRGBPack()

Parameters

result — A 16-bit or 32-bit RGB color pel buffer to contain the generated color image. By default, *result* will have same dimensions as *red, green,* and *blue* pel buffers. You can allocate a larger *result*, but the processing speed of this function is accelerated if you pre-allocate *result* the same size as *red, green,* and *blue*.

red — An 8-bit grey scale pel buffer that contains the red color channel of an RGB color camera.

green — An 8-bit grey scale pel buffer that contains the green color channel of an RGB color camera.

blue — An 8-bit grey scale pel buffer that contains the blue color channel of an RGB color camera.

alpha — Not currently implemented. Specify 0 or leave the default value of 0 in place.

Throws

ccPelFunc::BadParams
Any of the source pel bufffers is unbound, or their image sizes do not match.

cfRGBSeparateColorPlanes()

```
#include <ch_cvl/rgbplane.h>

cfRGBSeparateColorPlanes();
```

Global function to separate 32-bit RGB pel buffers into three 8-bit grey scale pel buffers.

cfRGBSeparateColorPlanes

```
void cfRGBSeparateColorPlanes(
  const ccPelBuffer_const<ccPackedRGB32Pel> &src32,
  ccPelBuffer<c_UInt8> *dstRed,
  ccPelBuffer<c_UInt8> *dstGreen,
  ccPelBuffer<c_UInt8> *dstBlue);
```

Separates the 32-bit RGB color pel buffer src32 into three 8-bit greyscale buffers that correspond to each of the red, green, and blue color planes.

For best performance

- *src32* should be 8-byte aligned.
- *dstRed*, *dstGreen*, *dstBlue* should be 4-byte aligned.
- The width of all pel buffers should be multiple of 4.

This function does not allocate the destination pel buffers. You must allocate a greyscale pel buffer for each color plane you wish to extract from the color pel buffer.

Notes

This function is accelerated when run on a CPU that supports the MMX registers. However, MMX support is not required, the function runs on any CPU supported by CVL.

CPUs that support the MMX registers include the Intel Pentium II, Pentium III, Pentium 4, Xeon, and Celeron processors. Some but not all members of the original Pentium family included MMX support.

cfRGBSeparateColorPlanes()

Parameters

src32 A 32-bit RGB color pel buffer.

dstRed A pointer to a 8-bit grey scale pel buffer with the same dimensions as *src32* to hold the red plane of the color pel buffer. If null, no color information for this plane is produced.

dstGreen A pointer to a 8-bit grey scale pel buffer with the same dimensions as *src32* to hold the green plane of the color pel buffer. If null, no color information for this plane is produced.

dstBlue A pointer to a 8-bit grey scale pel buffer with the same dimensions as *src32* to hold the blue plane of the color pel buffer. If null, no color information for this plane is produced.

Throws

ccPel::UnboundWindow

src32, dstRed, dstGreen, or *dstBlue* is unbound.

ccPel::BadWindow

One of the target pel buffers is bound but not the same size as the *src32*; or the three destination pel buffers do not have the same pitch.

Example

```
ccPelBuffer<ccPackedRGB32Pel> source32;
ccPelBuffer<c_Uint8> red(w,h), green(w,h), blue(w,h);
// w and h are the dimensions of the acquired images

// ...
// acquire 32-bit color image data into source32
// ...

// Separate source32 into all three color planes
cfRGBSeparateColorPlanes(source32, &red, &green, &blue);
```

cfSampleConvolve()

```
#include <ch_cvl/smplconv.h>

cfSampleConvolve();
```

Global function that performs smoothing and downsampling on the supplied 8-bit or 16-bit source image.

cfSampleConvolve

```
void cfSampleConvolve(
  const ccPelBuffer_const<c_UInt8> &src,
  const ccSampleConvolveParams &params,
  ccPelBuffer<c_UInt8> &dst);

ccPelBuffer<c_UInt8> cfSampleConvolve(
  const ccPelBuffer_const<c_UInt8> &src,
  const ccSampleConvolveParams &params);

void cfSampleConvolve(
  const ccPelBuffer_const<c_UInt16> &src,
  const ccSampleConvolveParams &params,
  ccPelBuffer<c_UInt16> &dst);

ccPelBuffer<c_UInt16> cfSampleConvolve(
  const ccPelBuffer_const<c_UInt16> &src,
  const ccSampleConvolveParams &params);
```

- ```
  void cfSampleConvolve(
    const ccPelBuffer_const<c_UInt8> &src,
    const ccSampleConvolveParams &params,
    ccPelBuffer<c_UInt8> &dst);
  ```

  Performs smoothing and downsampling on the supplied 8-bit image.

  Notes

  Non-integer *sampleStep* values are fully supported, but generally run slower than **FLOOR[***sampleStep***]** and much slower than **CEIL[***sampleStep***]**.

  The destination image respects client coordinates. The relationship of image and client coordinates of the result is determined by the sample.

  A default-constructed **ccSampleConvolveParams** object cannot be used to call this function. A valid kernel must be set, either directly with the **ccSampleConvolveParams::kernelX()** and **ccSampleConvolveParams::kernelY()** functions, or by using **ccSampleConvolveParams::setGaussSample()** or **ccSampleConvolveParams::setGaussSmoothing()**.
  ```

Parameters

src The 8-bit source image you supply.

params The sample convolve parameters.

dst The destination image.

Throws

ccPel::UnboundWindow
src is unbound.

ccPelFunc::Overlap
Source and destination images share pixels.

ccSampleConvolveDefs::BadParams
The (internally stored calculated or provided) kernelX_ or kernelY_ value has not been set (or both).

- ```
  ccPelBuffer<c_UInt8> cfSampleConvolve(
    const ccPelBuffer_const<c_UInt8> &src,
    const ccSampleConvolveParams &params);
  ```

  Performs smoothing and downsampling on the supplied 8-bit image. The functionality of this overload is the same as that of the first overload, with the difference that it does not return the destination image as an argument but as its return value.

  Parameters

  src The 8-bit source image you supply.

  params The sample convolve parameters.

- ```
  void cfSampleConvolve(
    const ccPelBuffer_const<c_UInt16> &src,
    const ccSampleConvolveParams &params,
    ccPelBuffer<c_UInt16> &dst);
  ```

  Performs smoothing and downsampling on the supplied 16-bit image. The functionality of this overload is the same as that of the first overload.

  Parameters

  src The 16-bit source image you supply.

  params The sample convolve parameters.

  dst The destination image.

- ```
  ccPelBuffer<c_UInt16> cfSampleConvolve(
    const ccPelBuffer_const<c_UInt16> &src,
    const ccSampleConvolveParams &params);
  ```

  Performs smoothing and downsampling on the supplied 16-bit image. The functionality of this overload is the same as that of the first overload, with the difference that it does not return the destination image as an argument but as its return value.

  Parameters

src	The 16-bit source image you supply.
params	The sample convolve parameters.
  ```

cfSampleConvolve()

cfSampledImageWarp()

```
#include <ch_cvl/simgwrp.h>

cfSampledImageWarp();
```

A global function that fills a destination image with grey values from a source image using a source image-from-destination image transform, and a destination image sampling rate.

cfSampledImageWarp

```
void cfSampledImageWarp(
  const ccPelBuffer_const<c_UInt8>& srcImage,
  ccPelBuffer<c_UInt8>& dstImage,
  const cc2XformBasePtrh_const& srcImageFromDstImage,
  c_Int32 samplingRate);
```

This function fills *dstImage* with grey values from *srcImage* using the *srcImageFromDstImage* transform. Each point in *destImage* is mapped by the *srcImageFromDstImage* transform to obtain locations in the *srcImage*. The bilinearly interpolated grey values associated with these positions are then assigned to their corresponding locations in *dstImage*.

cfSampledImageWarp() does not map every pixel in the *dstImage* with *dstImageFromSrcImage*. Instead, *dstImage* is sampled at a rate equal to the *sampingRate* (specified in image coordinates) in both its x- and y-directions. The mapped locations of all other pixels are then inferred from these mapped points. The *samplingRate* controls both the mapping accuracy and speed where smaller values correspond to higher accuracy at the cost of longer run times.

Notes

If points within *dstImage* map to locations outside of *srcImage*, a grey-value of 0 is used.

The *clientFromImageXform* transformation of *dstImage* is computed by the following:

(clientFromImageXform of dst) =
(clientFromImageXform of src)*(srcFromDstTransform provided)

srcImage and *dstImage* must not share pixels.

Parameters

srcImage The source image.

dstImage The destination image.

srcImageFromDstImage
The source image from destination image transform.

samplingRate The destination image sampling rate.

Throws

ccSampledImageWarpDefs::BadParams
If *samplingRate* < 1.

ccPel::UnboundWindow
If *srcImage* or *dstImage* are unbound.

ccPelFunc::Overlap
If *srcImage* and *dstImage* overlap.

cfSegmentColorImage

```
#include <ch_cvl/colsegm.h>
```

```
cfSegmentColorImage()
```

The Color Segmenter tool takes a vector of **ccColorRange**s and a color image, and segments the color image to produce a greyscale image in which the light pixels correspond to the colors in the color image that matched the color ranges.

Each **ccColorRange** value defines three fuzzy segmentation functions based on the nominal value and tolerances for the three color planes.

For each **ccColorRange** value, the Color Segmenter tool computes an intermediate image for each color plane using the corresponding fuzzy function, computes the product of the three images, and scales the values from 0 to 255. The final result is the maximum of all the segmentation images for the **ccColorRange** values in the vector.

cfSegmentColorImage

```
ccPelBuffer<c_UInt8> cfSegmentColorImage(
                const cc3PlanePelBuffer& colorImage,
                const cmStd vector<ccColorRange>& colors);

void cfSegmentColorImage(
                const cc3PlanePelBuffer& colorImage,
                const cmStd vector<ccColorRange>& colors,
                ccPelBuffer<c_UInt8>& result);
```

- ```
  ccPelBuffer<c_UInt8> cfSegmentColorImage(
                  const cc3PlanePelBuffer& colorImage,
                  const cmStd vector<ccColorRange>& colors);
  ```

  Segments the color image according to the collection of reference colors, and returns the segmentation in a result image.

  Parameters

  colorImage The image to segment.

  colors The reference colors.

  Throws

  ccColorSegmenterDefs::BadParams
  the size of *colors* is zero

  ccColorSegmenterDefs::BadImage
  colorImage is not bound
  ```

ccColorSegmenterDefs::BadColorSpace
colorImage's color space is **ccColorSpaceDefs::eUnknown** or an illegal value

ccColorSegmenterDefs::BadColorSpace
colors has a color with color space **ccColorSpaceDefs::eUnknown**.

ccColorSegmenterDefs::BadColorSpace
colorImage cannot be converted to the color space of one of the color ranges.

- ```
  void cfSegmentColorImage(
                  const cc3PlanePelBuffer& colorImage,
                  const cmStd vector<ccColorRange>& colors,
                  ccPelBuffer<c_UInt8>& result);
  ```

Segments the color image according to the collection of reference colors, and stores the result of the segmentation in *result*.

If *result* is not bound, it is allocated to be the same size as *colorImage*. If *result* is bound, only the area of *colorImage* and *result* that overlap is processed.

Parameters

colorImage The image to segment.

colors The reference colors.

result The grey scale pel buffer that holds the results.

Throws

ccColorSegmenterDefs::BadParams
the size of *colors* is zero

ccColorSegmenterDefs::BadImage
colorImage is not bound

ccColorSegmenterDefs::BadColorSpace
colorImage's color space is **ccColorSpaceDefs::eUnknown** or an illegal value

ccColorSegmenterDefs::BadColorSpace
colors has a color with color space **ccColorSpaceDefs::eUnknown**.
  ```

ccColorSegmenterDefs::BadColorSpace
colorImage cannot be converted to the color space of one of the color ranges.

cfSegmentColorImage

cfSegmentFeature()

```
#include <ch_cvl/pmifproc.h>

cfSegmentFeature();
```

Global function to segment a PatInspect feature into feature segments.

cfSegmentFeature

```
template <class BoundaryPoint>
cmStd vector<ccFeatureSegment> cfSegmentFeature(
  const cc_PMInspectFeature<BoundaryPoint>& feature,
  const ccRadian& maxAngle, c_Int32 minLength);
```

Segments the supplied **cc_PMInspectFeature** into one or more **ccFeatureSegment**s.

Parameters

feature	A PatInspect feature.
maxAngle	The maximum angle between adjacent boundary points permitted within a feature segment. *feature* will be segmented at boundary points with angles greater than *maxAngle*.
minLength	The minimum number of boundary points in a feature segment. Feature segments with fewer than *minLength* boundary points are discarded.

cfSegmentFeature()

cfSemiTrigger()

```
#include <ch_cvl/trigmodl.h>

cfSemiTrigger();
```

Global function to retrieve a semi trigger model object reference.

cfSemiTrigger

```
const ccTriggerModel& cfSemiTrigger();
```

When you specify **cfSemiTrigger()** as the trigger model, starting an acquisition is a two-step process:

1. You first prepare your application for acquisition by calling **ccAcqFifo::start()**. Each call to **ccAcqFifo::start()** prepares the hardware for one acquisition when there is a transition on the trigger input line.
2. The next transition on the trigger input line initiates the acquisition.

Notes

If **ccTriggerProp::triggerEnable()** is set to false, transitions on the trigger input line are ignored, and calls to **ccAcqFifo::start()** cause acquisition preparations to queue. **ccTriggerProp::triggerEnable()** must be true in order for acquisitions to proceed.

The default behavior is for a low-to-high transition on the trigger input line to trigger an acquisition. You can change this behavior to cause a high-to-low transition to trigger an acquisition by setting **ccTriggerProp::triggerLowToHigh()** to false.

cfSemiTrigger()

cfSetLanguage()

```
#include <ch_cvl/unicode.h>

cfSetLanguage();
```

Global function to select localized versions of resource DLLs to load. These DLLs are in the *\vision\bin\win32\cvl* directory and end with the three-letter code for the language. Cognex supplies only DLLs localized for US English (ENU).

There must be a complete set of resource files for each language. For example, if you want to localize only *cogstdENU.dll* to French (*cogstdFRA.dll*), you should make sure that you make copies of the other resource files and name them with the FRA suffix even if you do not localize them. If this function cannot find the complete set, it will throw an error.

For more information about localization, see *National Language Support* in the *Windows Base Services* section of the *Platform SDK* documentation.

cfSetLanguage

```
void cfSetLanguage(unsigned short languageId);
```

Selects which localized resource DLLs to load. If you do not call this function, CVL uses the US English resource DLLs whose names end in ENU.

You can call this function at any time to change languages while your application is running.

Parameters

languageId The language to use. Use the Windows **MAKELANGID()** macro to create a language ID.

Example

To use resource DLLs localized for Mexican Spanish (DLLs ending in ESM), you would call **cfSetLanguage()** like this:

```
cfSetLanguage(MAKELANGID(LANG_SPANISH,
                         SUBLANG_SPANISH_MEXICAN));
```

Throws

BadParams Localized versions of the resource DLLs for the language specified by *languageID* were not found. The *languageID* is set to its previous value.

cfSetLanguage()

cfSqr()

```
#include <ch_cvl/math.h>
```

```
cfSqr();
```

Global function that computes the square of a number (or a class that implements operator*).

cfSqr

```
template <class T> T cfSqr(T x);
```

Returns the square of the supplied value.

Parameters

T	The type of the value being squared.
x	The value to square.

cfSqr()

cfSetThreadPriority()

```
#include <ch_cvl/threads.h>

cfSetThreadPriority();
```

Global function that sets the thread priority of the current thread.

cfSetThreadPriority

```
void cfSetThreadPriority (cePriority newpri);
```

Parameters

newpri The new priority for this thread. *newpri* must be one of the following values:

cePriorityDefault
cePriorityIdle
cePriorityLowest
cePriorityBelowNormal
cePriorityNormal
cePriorityAboveNormal
cePriorityHighest
cePriorityTimeCritical

In CVL applications that will execute on the MVS-82400, do not raise the thread priority to *cePriorityTimeCritical*. If your application creates threads at this priority, your application on the embedded processor may experience problems with **ccTimer** objects and delays in image acquisition.

cfSetThreadPriority()

cfSlaveTrigger()

```
#include <ch_cvl/trigmodl.h>

cfSlaveTrigger();
```

Global function to retrieve a slave trigger model object reference.

cfSlaveTrigger

```
const ccTriggerModel& cfSlaveTrigger();
```

When you specify **cfSlaveTrigger()** as the trigger model, the acquisition FIFO becomes a slave to another acquisition FIFO. For each acquisition performed on a master FIFO, the slave FIFOs perform an acquisition at the same time. Master and slave FIFOs automatically coordinate with each other to ensure that all cameras capture images at the same time.

Master/slave FIFO configurations are also known as *synchronous* configurations. Configurations in which multiple FIFOs act independently of each other are known as *asynchronous* configurations.

In a synchronous configuration, a master FIFO can have more than one slave. The ability of a FIFO to act as master or slave depends on the frame grabber. Not all frame grabbers support the **cfSlaveTrigger()** trigger model, and those that do may have significant restrictions. For example, the FIFOs in a synchronous group must use consecutive camera ports, and the master FIFO must be associated with the first of those ports. Selecting an invalid synchronous group will not necessarily generate an exception, but will result in invalid results. Refer to the *Release Notes* for supported configurations.

Slaves cannot have other slaves. Use **ccTriggerProp::couldSlaveTo()** to check whether one FIFO can be slaved to another. **ccTriggerProp::couldSlaveTo()** indicates whether one FIFO can be a slave to another without taking into account current property settings. In particular, the master port property of the FIFOs must be set properly for synchronous configurations to work.

If a FIFO can be a slave to another FIFO, use **ccTriggerProp::triggerMaster()** to set the slave FIFO's master FIFO. Pointing **triggerMaster()** at another slave FIFO is not an error, and does not generate an exception, but the secondary slave will be unusable. A slave with an unbound **triggerMaster()** handle is also not considered an error, and is also effectively unusable.

If a synchronous configuration does not perform as expected, use **ccAcqFifo::isValid()** to check whether each FIFO is configured properly. If **isValid()** returns false, you can use the **ccAcqProblem** methods to determine the cause of the problem.

Notes

Starting with CVL 6.0, the **ccAcqProblem::hasInvalidMaster()** and **hasInvalidSlave()** methods are provided for backward compatibility only. Cognex recommends using other members of the **ccAcqProblem** class to obtain more

specific indications of which setup error was detected. For example, **slavePortInvalid()** says whether the camera port settings are valid, and **slaveNotSlaveTrigger()** says whether a FIFO has been assigned a trigger master but has not been assigned a slave trigger model. See the **ccAcqProblem** class for details.

Notes

If you call **ccAcqFifo::start()** when the slave trigger model is selected, your application will throw *ccAcqFifo::StartNotAllowed*.

Neither the **ccTriggerProp::triggerEnable()** nor the **ccTriggerProp::triggerLowToHigh()** property applies for this trigger model.

cfSystemTimeGet()

```
#include <ch_cvl/systime.h>

cfSystemTimeGet();
```

Global function to obtain the current system date and time.

cfSystemTimeGet

```
void cfSystemTimeGet(csSystemTime& time);
```

Obtains the current system date and time. When called from an application running on the host system, this function obtains the host system time.

The time is expressed in Coordinated Universal Time (UTC).

Parameters

time The current system time is placed in the supplied **csSystemTime** structure:

```
typedef struct
{
c_Int16 year; /* 1999, 2000 ... */
c_Int16 month; /* Jan = 1, Feb = 2 ... */
c_Int16 dayOfWeek; /* Sun = 0, Mon = 1 ... */
c_Int16 day; /* The day of the month. */
c_Int16 hour; /* The current hour (0-23). */
c_Int16 minute; /* The current minute (0-59). */
c_Int16 second; /* The current second (0-59). */
c_Int16 millisecond; /* Current msec (0-999) */
} csSystemTime;
```

cfSystemTimeGet()

cfSystemTimeSet()

```
#include <ch_cvl/systime.h>

cfSystemTimeSet();
```

Global function to set the current system date and time.

cfSystemTimeSet

```
bool cfSystemTimeSet(const csSystemTime& time);
```

Sets the current system date and time. When called from an application running on the host system, this function sets the host system time.

The time is expressed in Coordinated Universal Time (UTC).

This function returns a nonzero value to indicate success, zero to indicate failure.

Parameters

time The system time is set to the values in the supplied **csSystemTime** structure:

```
typedef struct
{
c_Int16 year; /* 1999, 2000 ... */
c_Int16 month; /* Jan = 1, Feb = 2 ... */
c_Int16 dayOfWeek; /* Sun = 0, Mon = 1 ... */
c_Int16 day; /* The day of the month. */
c_Int16 hour; /* The current hour (0-23). */
c_Int16 minute; /* The current minute (0-59). */
c_Int16 second; /* The current second (0-59). */
c_Int16 millisecond; /* Current msec (0-999) */
} csSystemTime;
```

Throws

ccSysTimeBadArgument::BadParams
One or more of the members of *time* is invalid.

Notes

The *dayOfWeek* member of *time* is ignored. The *year* member of *time* must be greater than or equal to 1980.

cfSystemTimeSet()

cfThreadCleanup()

```
#include <ch_cvl/threads.h>

cfThreadCleanup();
```

Global function that must be called at the termination of all threads that both

- Call CVL functions

  and

- Were not created using **cfCreateThread()**

You do not need to call this function for any threads that do not meet both of the requirements listed above.

This function does not need to be called for a program's main thread.

cfThreadCleanup

```
void cfThreadCleanup();
```

Caution

Once this function has been called within a thread, no further CVL calls can be made by that thread. This restriction includes the destructors for CVL objects. Accordingly, if you create a thread that creates any CVL objects, those objects must be destroyed before the thread calls **cfThreadCleanup()**.

If your thread creates CVL objects on the stack, you need to include an extra scope within your thread function, as shown below:

```
myThreadProc(void *arg)
{
  {
  ccPelBuffer<c_UInt8> myImage;

    ...

  } // End of extra scope -- myImage destroyed here

// Now safe to call cfThreadCleanup()
cfThreadCleanup()
}
```

cfThreadCleanup()

cfThresholdWGV()

```
#include <ch_cvl/thresh.h>

cfThresholdWGV();
```

Global function to calculate the threshold for a histogram by minimizing the within-group variance. This technique is particularly useful for obtaining a threshold value for noisy images, saturated images, or multimodal histograms. You typically determine the threshold before segmenting an image with the Blob tool.

cfThresholdWGV

```
void cfThresholdWGV(
  const cmStd vector<c_UInt32>& histogram,
  ccThresholdResult& result,
  c_Int32 leftIndex = 0,
  c_Int32 rightIndex = -1);
```

Computes the threshold for the specified range of bins by minimizing the within-group variance.

Parameters

histogram The histogram for which to compute a threshold. *histogram* must have a size greater than 0.

result A threshold result object.

leftIndex The index of the left pixel bin for the partial histogram. *leftIndex* is included in the threshold computation.

rightIndex The index of the right pixel bin for the partial histogram. *rightIndex* is excluded from the threshold computation. If you specify -1, **histogram.size()** is used as the value of *rightIndex*.

Throws

ccThresholdDefs::BadParams

leftIndex is less than 0; *rightIndex* is not -1 and is greater than the value returned by **histogram.size()**; *rightIndex* is not -1 and is less than or equal to *leftIndex*; or the partial histogram does not contain any nonzero values.

Notes

The within-group variance technique for calculating a threshold applies only to those partial histograms that have at least two nonzero bin values between *leftIndex* and *rightIndex*. In this case, **cfThresholdWGV()** calculates a threshold that lies between the indices and reports a nonzero score. The threshold computation includes the *leftIndex* value but excludes the *rightIndex* value. The two groups of bins created by the threshold have indices in the range *leftIndex*, inclusive, through

the threshold, exclusive, and the threshold, inclusive, through *rightIndex*, exclusive. When several possible thresholds minimize the within-group variance, **cfThresholdWGV()** returns the minimum threshold value.

If the histogram contains only one nonzero value, the within-group variance technique cannot be used. **cfThresholdWGV()** returns the index of the nonzero value and a score of 0.

cfTruePeak()

```
#include <ch_cvl/edge.h>

cfTruePeak();
```

Global function that performs whole-pixel peak detection on the supplied edge magnitude and edge angle images.

cfTruePeak

```
template<class T>
ccPelBuffer<T> cfTruePeak(
  const ccPelBuffer_const<T>& magImage,
  const ccPelBuffer_const<c_UInt8>& angImage,
  T magThreshold, c_Int32* pNPeaks = 0);

template<class T>
void cfTruePeak(const ccPelBuffer_const<T>& magImage,
  const ccPelBuffer_const<c_UInt8>& angImage,
  ccPelBuffer<T>& dstImage, T magThreshold,
  c_Int32* pNPeaks = 0);
```

-
```
template<class T>
ccPelBuffer<T> cfTruePeak(
  const ccPelBuffer_const<T>& magImage,
  const ccPelBuffer_const<c_UInt8>& angImage,
  T magThreshold, c_Int32* pNPeaks = 0);
```

Performs whole-pixel peak detection on the supplied edge magnitude and edge angle images and returns an edge magnitude image where pixels which are true edge peaks are set to the magnitude of the pixels in the input image, and all other pixels are set to 0.

For each pixel in *magImage* with a nonzero value and with a corresponding pixel in *angImage*, the Edge tool applies an angle-sensitive 3x1 neighborhood operator. Only those pixels which have a greater magnitude than both neighbors are included in the returned edge peak image.

Parameters

T	Template parameters specifying the type of the magnitude image and threshold. *T* must be of type **c_Uint8**.
magImage	The input edge magnitude image
angImage	The input edge angle image
magThreshold	A magnitude threshold. All pixels in *magImage* with a value less than *magThreshold* are discarded.
pNPeaks	If *pNPeaks* is not *NULL*, the total number of true peaks is written to the **c_Int32** pointed to by *pNPeaks*.

- ```
  template<class T>
  void cfTruePeak(const ccPelBuffer_const<T>& magImage,
    const ccPelBuffer_const<c_UInt8>& angImage,
    ccPelBuffer<T>& dstImage, T magThreshold,
    c_Int32* pNPeaks = 0);
  ```

  Performs whole-pixel peak detection on the supplied edge magnitude and edge angle images and returns an edge magnitude image where pixels which are true edge peaks are set to the magnitude of the pixels in the input image, and all other pixels are set to 0.

  For each pixel in *magImage* with a nonzero value and with a corresponding pixel in *angImage*, the Edge tool applies an angle-sensitive 3x1 neighborhood operator. Only those pixels which have a greater magnitude than both neighbors are included in the returned edge peak image.

  Parameters

T	Template parameters specifying the type of the magnitude image and threshold. *T* must be of type **c_Uint8**.
magImage	The input edge magnitude image
angImage	The input edge angle image
dstImage	The image into which the true edge peak output image is placed
magThreshold	A magnitude threshold. All pixels in *magImage* with a value less than *magThreshold* are discarded.
pNPeaks	If *pNPeaks* is not *NULL*, the total number of true peaks is written to the **c_Int32** pointed to by *pNPeaks*.
  ```

cfVerifyOcrChecksum()

```
#include <ch_cvl/acuread.h>

cfVerifyOcrChecksum();
```

Global function to verify that a string passes a checksum test.

cfVerifyOCRChecksum

```
bool cfVerifyOcrString(
   ccAcuReadDefs::Checksum checksum,
   const ccCvlString &str);
```

Returns true if the supplied string has the supplied checksum.

Parameters

checksum The checksum to verify. *checksum* must be one of the following values:

ccAcuReadDefs::eSemi
ccAcuReadDefs::eBC412
ccAcuReadDefs::eIBM412

str The string to validate.

Throws

ccAcuReadDefs::BadParams
The length of *str* is greater than *ccAcuReadDefs::kMaxStringLength*, *str* contains invalid characters, or *checksum* is *ccAcuReadDefs::eNone* or *ccAcuReadDefs::eVirtual*.

cfVerifyOcrChecksum()

cfVerifyOcrString()

```
#include <ch_cvl/acuread.h>

cfVerifyOcrString();
```

Global function to verify that a string is valid given a fielding string.

cfVerifyOCRString

```
bool cfVerifyOcrString(
   const cmStd vector<cccCvlString> &fieldMap,
   const ccCvlString &str);
```

Returns true if each character in *str* is one of the characters in the corresponding location in *fieldMap*, false otherwise.

Parameters

fieldMap A vector of **ccCvlString**s. Each vector contains all of the valid characters for that field position.

str The string to validate.

Throws

ccAcuReadDefs::BadParams
The size of *fieldMap* is greater than *ccAcuReadDefs::kMaxStringLength* + 1, the length of *str* is greater than *ccAcuReadDefs::kMaxStringLength*, the length of any of the strings in *fieldMap* is greater than *ccAcuReadDefs::kMaxFieldStringLength*, or *fieldMap* contains invalid characters.

cfVerifyOcrString()

cfWaitForContinue()

```
#include <ch_cvl/gui.h>

cfWaitForContinue();
```

Global function that displays a dialog box containing **Continue** and **Cancel** buttons on the host computer system, then waits for the user to click one of the two buttons.

cfWaitForContinue

```
bool cfWaitForContinue();

bool cfWaitForContinue(int xPos, int yPos);
```

- `bool cfWaitForContinue();`

  Displays the alert box in the center of the screen. Returns true if the user clicks **Continue** and false if he clicks **Cancel**.

- `bool cfWaitForContinue(int xPos, int yPos);`

  Displays the alert box at the specified location. Returns true if the user clicks **Continue** and false if he clicks **Cancel**.

  Parameters

xPos	The x-location of the upper-left corner of the alert box in host display pixels.
yPos	The y-location of the upper-left corner of the alert box in host display pixels.

cfWaitForContinue()

cfWaitForThreadTermination()

```
#include <ch_cvl/threads.h>

cfWaitForThreadTermination();
```

Global function that blocks until the supplied thread terminates.

cfWaitForThreadTermination

```
void cfWaitForThreadTermination(ccThreadID &thread);
```

Parameters

thread	The thread to wait for.

cfWaitForThreadTermination()

CompleteArgs

```
#include <ch_cvl/acqbase.h>

class ccAcqFifo::CompleteArgs;
```

CompleteArgs is a nested class defined within the scope of **ccAcqFifo**.

This class is used as the argument to the **ccAcqFifo::completeAcq()** method. The design of this class allows emulation of named parameters, where only the arguments with non-default values need to be specified, in any order.

Notes

Using **ccAcqFifo::complete()** in derived classes to handle acquired images is now replaced by **ccAcqFifo::completeAcq()**. The older method is retained for backward compatibility, but use **ccAcqFifo::completeAcq()** for all new code.

Class Properties

Copyable	No
Derivable	No
Archiveable	No

Constructors/Destructors

CompleteArgs

```
explicit inline CompleteArgs();
```

Constructs a **CompleteArgs** object that can be passed to **ccAcqFifo::completeAcq()** on an acquisition FIFO, or passed to an acquisition callback function. If not overridden, the arguments are initialized with the default values shown in this table:

Argument	Default Value
makeLocal()	true
maxWait()	HUGE_VAL
startReqStatus()	None
acquireInfo()	None
autoStart()	false

Public Member Functions

The member functions of **ccAcqFifo::CompleteArgs()** are used as if they were parameters to an invocation of **completeAcq()**. You only need to specify a parameter when changing a default value. If you wish to use a **ccAcquireInfo** object to contain the results of an image acquisition, you must specify the **acquireInfo()** parameter. If you wish to check the **start()** completion status, you must specify the **startReqStatus()** parameter.

There are two ways to specify the arguments to **completeAcq()** using a **CompleteArgs** object, the long form and the shortcut form. Cognex documentation shows the shortcut form in code examples, and recommends using it.

In the long form method, you instantiate a **CompleteArgs** object and use its member functions, one at a time, to override the default-constructed value for any argument you want to change, as shown in the following example.

```
ccAcquireInfo info;
ceStartReqStatus status;

ccAcqFifo::CompleteArgs args;

args.acquireInfo(&info);
args.startReqStatus(&status);
args.maxWait(5.0);
args.makeLocal(false);

ccPelBuffer<c_UInt8> pb = fifo->complete(args);
```

The shortcut form is made possible because all **CompleteArgs** member functions return a reference to the **CompleteArgs** object. For this reason, you can chain each member function off the constructor, resulting in a simple list of white-space separated **CompleteArgs** arguments, as shown in the following example. This example accomplishes exactly the same as the preceding example.

```
ccAcquireInfo info;
ceStartReqStatus status;
ccAcqImagePtrh img = fifo->completeAcq
          (ccAcqFifo::CompleteArgs().acquireInfo(&info)
                                    .startReqStatus(&status)
                                    .maxWait(5.0)
                                    .makeLocal(false));
```

The four parameters can be specified in any order.

The argument chaining makes *args* redundant, and it can be dropped. Using the shortcut form, all function-call arguments to **completeAcq()** are consolidated into a single statement.

As another example, if you want only to specify a **ccAcquireInfo** object, set **makeLocal()** false, and leave the other arguments in their default values, use code like the following:

```
ccAcquireInfo info;
ccPelBuffer<c_UInt8> pb = fifo->complete
          (ccAcqFifo::CompleteArgs().makeLocal(false)
                                    .acquireInfo(&info));
```

acquireInfo

```
ccAcquireInfo* acquireInfo() const;
CompleteArgs& acquireInfo();
```

- `ccAcquireInfo* acquireInfo() const;`

  Returns the **ccAcquireInfo** object, if one was set.

- `CompleteArgs& acquireInfo();`

  Specifies the use of a **ccAcquireInfo** object to contain the results of the image acquisition. **ccAcquireInfo** is a class that stores several pieces of information about each image acquisition. See **ccAcquireInfo** on page 261.

  Use this function as shown in this example:

```
ccAcquireInfo info;
ccPelBuffer<c_UInt8> pb = fifo->complete
(ccAcqFifo::CompleteArgs().acquireInfo(&info));
```

makeLocal

```
bool makeLocal() const;
CompleteArgs& makeLocal();
```

- `bool makeLocal() const;`

  Returns the current setting of the **makeLocal()** parameter.

- `CompleteArgs& makeLocal();`

  Sets the Boolean *makeLocal* parameter; the default value is true. Use this function as shown in this example:

```
ccPelBuffer<c_UInt8> pb = fifo->complete
          (ccAcqFifo::CompleteArgs().makeLocal(false));
```

Notes

The **makeLocal()** parameter is used in PC-based frame grabbers with the ability to store images on board. **makeLocal(true)** specifies that the image is to be automatically copied to PC memory. **makeLocal(false)** specifies that the image is to be left on the frame grabber. If the image is left on the board, it can be discarded, transferred to PC memory at a later time, or transferred directly to the PC's display adapter for display.

maxWait

```
double maxWait() const;

CompleteArgs& maxWait();
```

- `double maxWait() const;`

  Returns the current value of the **maxWait()** parameter.

- `CompleteArgs& maxWait();`

  Sets the **maxWait()** parameter, specified in seconds; the default value is HUGE_VAL. Use this function as shown in this example:

```
ccPelBuffer<c_UInt8> pb = fifo->complete
          (ccAcqFifo::CompleteArgs().maxWait(5.0));
```

Notes

The **maxWait()** parameter is the maximum number of seconds to wait for a completed image acquisition to become available. The special value HUGE_VAL means to wait indefinitely. If the time specified by **maxWait()** expires, **complete()** returns an unbound pel buffer, and **ccAcquireInfo()->failure().isIncomplete()** is true.

startReqStatus

```
ccAcqFifo::ceStartReqStatus* startReqStatus() const;

CompleteArgs& startReqStatus();
```

- `ccAcqFifo::ceStartReqStatus* startReqStatus() const;`

  Returns the current setting of the **ceStartReqStatus** parameter.

- `CompleteArgs& startReqStatus();`

  Specifies a pointer to **ccAcqFifo::ceStartReqStatus** to contain the **start()** completion status. Use this function as shown in this example:

```
ccAcqFifo::ceStartReqStatus status;
ccPelBuffer<c_UInt8> pb = fifo->complete

(ccAcqFifo::CompleteArgs().startReqStatus(&status));
```

Notes

If **startReqStatus()** is non-null, then upon acquisition completion, it contains the status of the **ccAcqFifo::start()** call that initiated the acquisition. This determines whether an additional call to **start()** is necessary to obtain the next image, whether the *appTag* is valid, and whether the FIFO is in sequence with the other FIFOs of a master-slave sequence. See the description of **ccAcqFifo::ceStartReqStatus()** on page 218 for further explanation.

---

autoStart

```
bool autoStart() const;

CompleteArgs& autoStart();
```

- `bool autoStart() const;`

  Returns the current setting of the **autoStart()** parameter.

- `CompleteArgs& autoStart();`

  Deprecated function, for backwards compatibility only.

Notes

If **autoStart()** is true and the current trigger model allows **start()** to be invoked, then **start()** is automatically invoked following this acquisition's completion, with an *appTag* of zero. If the acquisition was incomplete because the *maxWait* parameter timed out, (as reported by **ccAcquireInfo()->failure().isIncomplete()**) or if this function throws, **start()** is not invoked.

CompleteArgs

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Index