Open eVision

Code Identification Tools

Terms of Use

Contents

1. Dealing with Pixel Containers and Files

1.1. Pixel Container Definition

Images

Open eVision image objects contain image data that represents rectangular images.

Each image object has a data buffer, accessible via a pointer, where pixel values are stored contiguously, row by row.
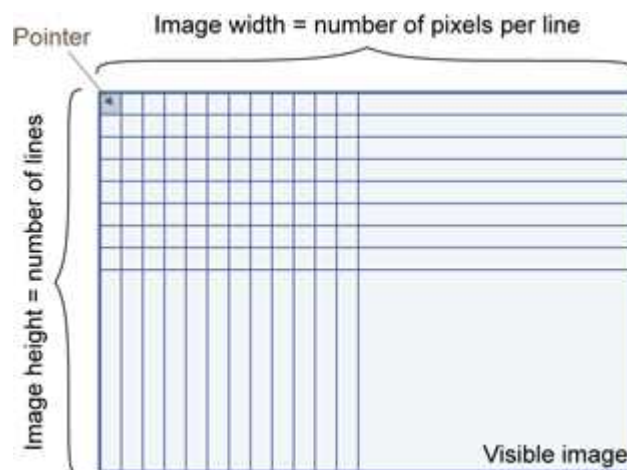
Image main parameters

An Open eVision image object has a rectangular array of pixels characterized by `EBaseROI` parameters .

- `Width` is the number of columns (pixels) per row of the image.
- `Height` is the number of rows of the image. (Maximum width / height is 32,767 (2^{15}-1) in Open eVision 32-bit, and 2,147,483,647 (2^{31}-1) in Open eVision 64-bit.)
- `Size` is the width and height.

The `Plane` parameter contains the number of color components. Gray-level images = 1. Color images = 3.

Classes

Image and ROI classes derive from abstract class `EBaseROI` and inherit all its properties.
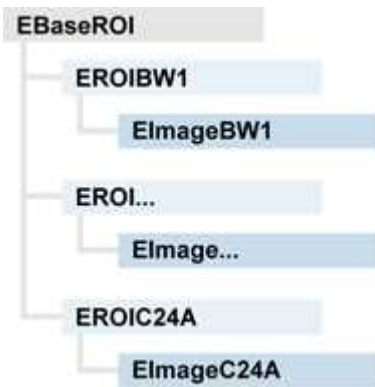
Depth maps

A depth map is way to represent a 3D object using a 2D grayscale image, each pixel in the image representing a 3D point.

The pixel coordinates are the representation of the X and Y coordinates of the point while the grayscale value of the pixel is a representation of the Z coordinate of the point.

Point clouds

A point cloud (https://en.wikipedia.org/wiki/Point_cloud) is an unstructured set of 3D points representing discrete positions on the surface of an object.
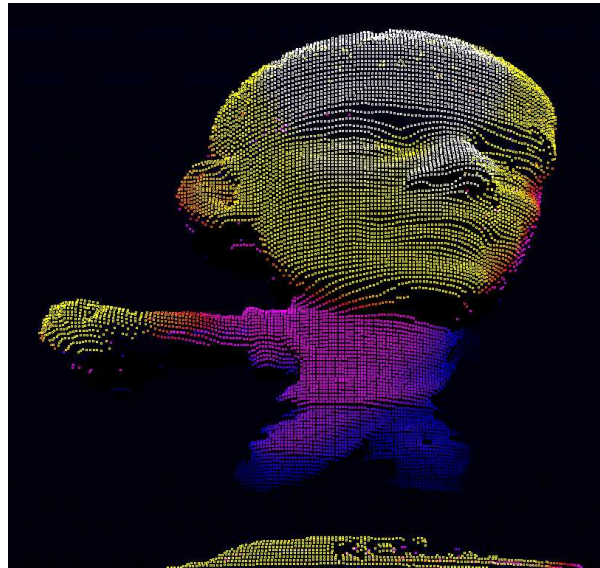
3D point clouds are produced by various 3D scanning techniques, such as Laser Triangulation, Time of Flight or Structured Lighting.

1.2. Pixel Container Types

Images

Several image types are supported according to their pixel types: black and white, gray levels, color, etc.

Easy.GetBestMatchingImageType returns the best matching image type for a given file on disk.

BW1	1-bit black and white images (8 pixels are stored in 1 byte)	`EImageBW1`
BW8	8-bit grayscale images (each pixel is stored in 1 byte)	`EImageBW8`
BW16	16-bit grayscale images (each pixel is stored in 2 bytes)	`EImageBW16`
BW32	32-bit grayscale images (each pixel is stored in 4 bytes)	`EImageBW32`
C15	15-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB15 color images and MultiCam RGB15 format.	`EImageC15`

C16	16-bit color images (each pixel is stored in 2 bytes). Compatible with Microsoft® Windows RGB16 color images and MultiCam RGB16 format.	`EImageC16`
C24	C24 images store 24-bit color images (each pixel is stored in 3 bytes). Compatible with Microsoft® Windows RGB24 color images and MultiCam RGB24 format.	`EImageC24`
C24A	C24A images store 32-bit color images (each pixel is stored in 4 bytes). Compatible with Microsoft® Windows RGB32 color images and MultiCam RGB32 format.	`EImageC24A`

Depth Maps

8 and 16-bit depth map values are stored in buffers compatible with the 2D Open eVision images.

EDepth8	8-bit depth map (each pixel is stored in 1 byte as an integer)	`EDepthMap8`
EDepth16	16-bit depth map (each pixel is stored in 2 bytes as a fixed point)	`EDepthMap16`
EDepth32f	32-bit depth map (each pixel is stored in 4 bytes as a float)	`EDepthMap32f`

Point Clouds

Point Cloud	Set of points coordinates (stored as float)	`E3DPointCloud`

1.3. Supported Image File Types

Type	Description
BMP	Uncompressed image data format (Windows Bitmap Format)
JPEG	Lossy data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 10918-1. Compression irretrievably looses quality.
JFIF	JPEG File Interchange Format

Type	Description
JPEG-2000	Data compression standard issued by the Joint Photographic Expert Group registered as ISO/IEC 15444-1 and ISO/IEC 15444-2. Open eVision supports only lossy compression format, file format and code stream variants. - **code stream** describes the image samples. - **file format** includes meta-information such as image resolution and color space.
PNG	Lossless data compression method (Portable Network Graphics).
Serialized	Euresys proprietary image file format obtained from the **serialization** of Open eVision image objects.
TIFF	Tag Image File Format is currently controlled by Adobe Systems and uses the LibTIFF third-party library to process images written for 5.0 or 6.0 TIFF specification. File **save** operations are lossless and use CCITT 1D compression for 1-bit binary pixel types and LZW compression for all others. File **load** operations support all TIFF variants listed in the LibTIFF specification.

1.4. Pixel and File Types Compatibility

Depth map to image conversion

For a 8- and 16-bit depth maps, the `AsImage()` method returns a compatible image object (respectively `EImageBW8` and `EImageBW16`) that can be used with Open eVision's 2D processing features.

Pixel and file types compatibility

Pixel access

The recommended method to access pixels is to use `SetImagePtr` and `GetImagePtr` to embed the image buffer access in your own code. See also Image Construction and Memory Allocation and Retrieving Pixel Values.

Use of the following methods should be limited because of the overhead incurred by each function call:

Direct access

`EROIBW8::GetPixel` and `SetPixel` methods are implemented in all image and ROI classes to read and write a pixel value at given coordinates. To scan all pixels of an image, you could run a double loop on the X and Y coordinates and use `GetPixel` or `SetPixel` each iteration, but this is not recommended.

For performance reasons, these accessors should not be used when a significant number of pixel needs to be processed. When that is the case, retrieving the internal buffer pointer using `GetBufferPtr()` *and iterating on the pointer is recommended.*

Quick Access to BW8 Pixels

In BW8 images, a call to `EBW8PixelAccessor::GetPixel` or `SetPixel` will be faster than a direct `EROIBW8::GetPixel` or `SetPixel`.

Supported structures

- `EBW1`, `EBW8`, `EBW32`
- `EC15` (*), `EC16` (*), `EC24` (*)
- `EC24A`
- `EDepth8`, `EDepth16`, `EDepth32f`,

(*) These formats support RGB15 (5-5-5 bit packing), RGB16 (5-6-5 bit packing) and RGB32 (RGB + alpha channel) but they must be converted to/from EC24 using `EasyImage::Convert` before any processing.

> **Note:** *Transition with versions prior to eVision 6.5 should be seamless: image pixel types were defined using typedef of integral types, pixel values were treated as unsigned numbers and implicit conversion to/from previous types is provided.*

Pixel and File Type compatibility during Load or Save operations

Type	BMP	JPEG	JPEG2000	PNG	TIFF	Serialized
BW1	Ok	N/A	N/A	Ok	Ok	Ok
BW8	Ok	Ok	Ok	Ok	Ok	Ok
BW16	N/A	N/A	Ok	Ok	Ok (***)	Ok
BW32	N/A	N/A	N/A	N/A	Ok (***)	Ok
C15	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C16	Ok	Ok (**)	Ok (**)	Ok (**)	Ok (**)	Ok
C24	Ok	Ok	Ok	Ok	Ok (**)	Ok
C24A	Ok	N/A	N/A	Ok	N/A	Ok
Depth8	Ok	Ok	Ok	Ok	Ok	Ok
Depth16	N/A	N/A	Ok	Ok	Ok (***)	Ok
Depth32f	N/A	N/A	N/A	N/A	N/A	Ok

N/A: Not supported. An exception occurs if you use the combination.

Ok: Image integrity is preserved with no data loss (apart from JPEG and JPEG2000, lossy compression).

(**) C15 and C16 formats are automatically converted into C24 during the save operation.

(***) BW16 and BW32 are not supported by Baseline TIFF readers.

1.5. Color Types

EISH: Intensity, Saturation, Hue color system.

ELAB: CIE Lightness, a*, b* color system.

ELCH: Lightness, Chroma, Hue color system.

ELSH: Lightness, Saturation, Hue color system.

ELUV: CIE Lightness, u*, v* color system.

ERGB: NTSC/PAL/SMPTE Red, Green, Blue color system.

EVSH: Value, Saturation, Hue color system.

EXYZ: CIE XYZ color system.

EYIQ: CCIR Luma, Inphase, Quadrature color system.

EYSH: CCIR Luma, Saturation, Hue color system.

EYUV: CCIR Luma, U Chroma, V Chroma color system.

2. Manipulating Pixels Containers and Files

2.1. Pixel Container File Save

Images and Depth Maps

The `Save` method of an image or the `SaveImage` method of a depth map or a ZMap saves the image data of an image or of a depth map or a ZMap object into a file using two arguments:

- Path: path, filename, and file name extension.
- Image File Type. If omitted, the file name extension is used.

Images bigger than 65,536 (either width or height) must be saved in Open eVision proprietary format.

Save throws an exception when:

- The requested image file format is incompatible with the image pixel types
- The Auto file type selection method and the file name extension is not supported

When saving a 16-bit depth map, the fixed point precision is lost and the pixels are considered as 16-bit integers.

image file type arguments

Argument	Image File Type
EImageFileType_Auto(*)	Automatically determined by the filename extension. See below.
EImageFileType_Euresys	Open eVision Serialization.
EImageFileType_Bmp	Windows bitmap - BMP
EImageFileType_Jpeg	JPEG File Interchange Format - JFIF
EImageFileType_Jpeg2000	JPEG 2000 File format/Code Stream -JPEG2000
EImageFileType_Png	Portable Network Graphics - PNG
EImageFileType_Tiff	Tagged Image File Format - TIFF

(*) Default value.

Assigned image file type if argument is **ImageFileType_Auto** or missing

File name extension(*)	Automatically assigned image file type
BMP	Windows Bitmap Format
JPEG, JPG	JPEG File Interchange Format - JFIF
JP2	JPEG 2000 file format
J2K, J2C	JPEG 2000 Code Stream
PNG	Portable Network Graphics
TIFF, TIF	Tagged Image File Format

(*) Case-insensitive.

Saving JPEG and JPEG2000 lossy compressions

`SaveJpeg` and `SaveJpeg2K` specify the compression quality when saving compressed images. They have two arguments:

- Path: a string of characters including the path, filename, and file name extension.
- Compression quality of the image file, an integer value in range [0: 100].
  `SaveJpeg` saves image data using JPEG File Interchange Format – JFIF.
  `SaveJpeg2K` saves image data using JPEG 2000 File format.

JPEG compression values

JPEG compression	Description
JPEG_DEFAULT_QUALITY (-1)	Default quality (*)
100	Superb image quality, lowest compression factor
75	Good image quality (*)
50	Normal image quality
25	Average image quality
10	Bad Image quality

(*) The default quality corresponds to the good image quality (75).

Representative JPEG 2000 compression quality values

JPEG 2000 compression	Description
-1	Default quality (*)
1	Highest image quality, lowest compression factor

JPEG 2000 compression	Description
16	Good Image Quality (*) (16:1 rate)
512	Lowest image quality, highest compression factor

(*) The default quality corresponds to the good image quality (16:1 rate).

Point Clouds

- Use the `Save` method to save the point cloud in Open eVision proprietary file format.

- Use the `SavePCD` method to save the point cloud in a file compatible with other software such as PCL (Point Cloud Library).

2.2. Pixel Container File Load

Images and Depth Maps

- Use the `Load` method to load image data into an image object:

  □ It has one argument: the **path:** path, filename, and file name extension.

  □ File type is determined by the file format.

  □ The destination image is automatically resized according to the size of the image on disk.

- The `Load` method throws an exception when:

  □ File type identification fails

  □ File type is incompatible with pixel type of the image object

Serialized image files of Open eVision 1.1 and newer are incompatible with serialized image files of previous Open eVision versions.

When loading a BW16 image (with integer values) in a depth map, the fixed point precision set in the depth map (0 by default) is left unchanged and used.

Point Clouds

- Use the `Load` method to save the point cloud in Open eVision proprietary file format.

- Use the `LoadPCD` method to save the point cloud in a file compatible with other software such as PCL (Point Cloud Library).

2.3. Memory Allocation

An image can be constructed with an internal or external memory allocation.

Internal Memory Allocation

The image object dynamically allocates and unallocates a buffer. Memory management is transparent.
When the image size changes, re-allocation occurs.
When an image object is destroyed, the buffer is unallocated.
To declare an image with internal memory allocation:

1. Construct an image object, for instance `EImageBW8`, either with width and height arguments, OR using the `SetSize` function.

2. Access a given pixel. There are several functions that do this. `GetImagePtr` returns a pointer to the first byte of the pixel at given coordinates.

External Memory Allocation

The user controls buffer allocation, or links a third-party image in the memory buffer to an Open eVision image.
Image size and buffer address must be specified.
When an image object is destroyed, the buffer is unaffected.
To declare an image with external memory allocation:

1. Declare an image object, for instance `EImageBW8`.

2. Create a suitably sized and aligned buffer (see Image Buffer).

3. Set the image size with the `SetSize` function.

4. Access the buffer with `GetImagePtr`. See also Retrieving Pixel Values.

2.4. Image and Depth Map Buffer

Image and depth map pixels are stored contiguously, from top row to bottom, from left to right, in Windows bitmap format (top-down DIB[1]) into an associated buffer.

The buffer address is a pointer to the start address of the buffer, which contains the top left pixel of the image.

---

[1]device-independent bitmap
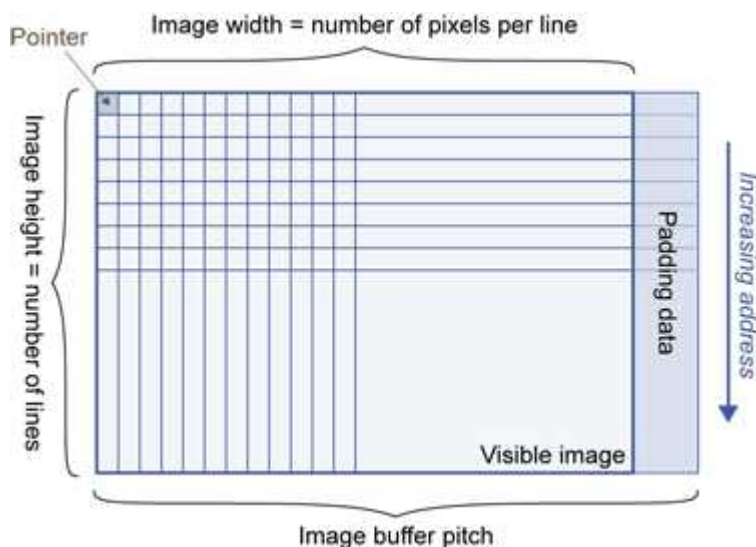
Image Buffer pitch

- Alignment must be a multiple of 4 bytes.

- Open eVision 1.2 onwards default pitch is 32 bytes for performance reasons (Open eVision 1.1.5 was 8 bytes).

Memory Layout

- `EImageBW1` stores 8 pixels in one byte.

    Example memory layout of the first 2 pixels of a BW1 image buffer:

- `EImageBW8` and `EDepthMap8` store each pixel in one byte.

    Example memory layout of the first pixels of a BW8 image buffer:
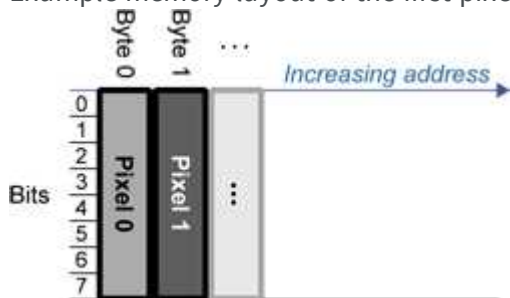
- `EImageBW16` stores each pixel in a 16-bit word (two bytes).

Example memory layout of the first pixels of a BW16 image buffer:
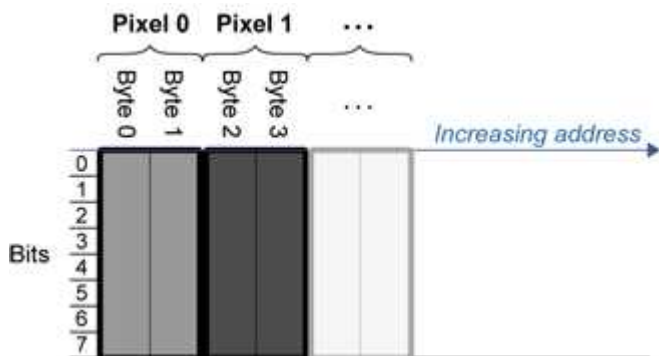
- `EImageC15` stores each pixel in 2 bytes. Each color component is coded with 5-bits.
  The 16th bit is left unused.

  Example memory layout of the first pixels of a C15 image buffer:
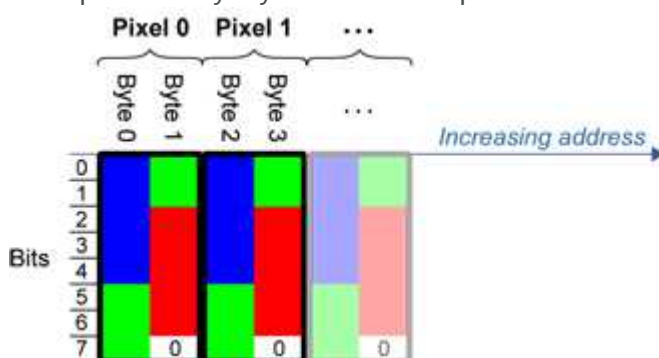
- `EImageC16` stores each pixel in 2 bytes. The first and third color components are coded with
  5-bits.
  The second color component is coded with 6-bits.

  Example memory layout of the first pixels of a C16 image buffer:
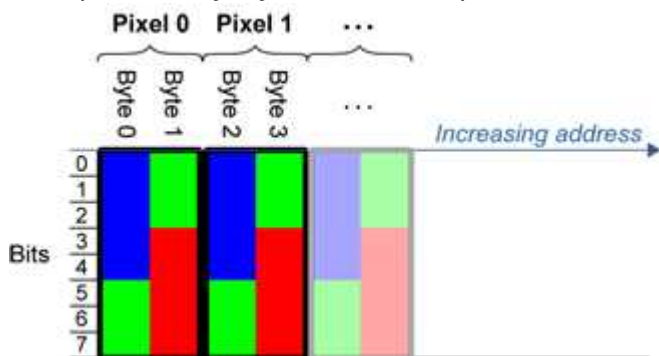
- `EDepthMap16` store each pixel in 2 bytes using a fixed point format.

- `EImageC24` stores each pixel in 3 bytes. Each color component is coded with 8-bits.

Example memory layout of the first pixels of a C24 image buffer:
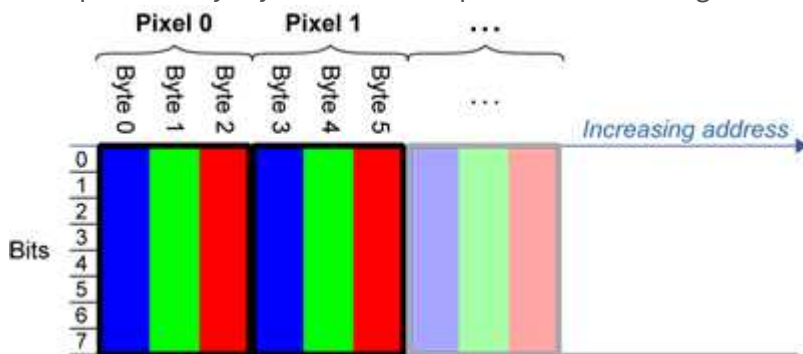
- ● `EImageC24A` stores each pixel in 4 bytes. Each color component is coded with 8-bits. The alpha channel is also coded with 8-bits.

  Example memory layout of the first pixels of a C24A image buffer:
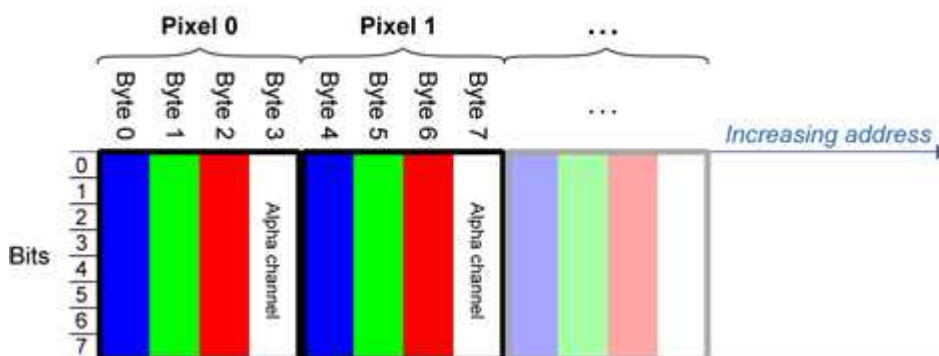
- ● `EDepthMap32f` store each pixel in 4 bytes using a float format.

2.5. Image Drawing and Overlay

- ■ Drawing uses Windows GDI[1] system calls.
  MFC[2] applications normally use `OnDraw` event handler to draw, where a pointer to a device context is available.
  Borland/CodeGear's OWL or VCL use a **Paint** event handler.

- ■ The color palette in 256-color display mode gives optimal rendering. Gray-level images can be improved using LUT[3]s (using histogram stretching techniques or pseudo-coloring).

- ■ The zoom can be different horizontally and vertically.

- ■ `DrawFrameWithCurrentPen` method draws a frame.

- ■ **Non-destructive overlaying** drawing operations do not alter the image contents, such as `MoveTo/LineTo`.

---

[1]Graphics Device Interface
[2]Microsoft Foundation Class
[3]LookUp Tables

- **Destructive overlaying** drawing operations alter the image contents by drawing inside the image such as `Easy::OpenImageGraphicContext`. Gray-level [color] images can only receive a gray-level [color] overlay.

2.6. 3D Rendering of 2D Images

These images are viewed by rotating them around the X-axis, then the Y-axis.

Gray 3D Rendering

`Easy::Render3D` prepares a 3-dimensional rendering where gray-level values are altitudes. Magnification factors in the three directions (X = width, Y = height and Z = depth) can be given. The rendered image appears as independent dots whose size can be adjusted to make the surface more or less opaque.
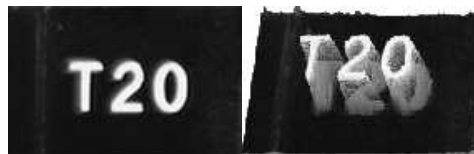
3D rendering

Color Histogram 3D Rendering

`Easy::RenderColorHistogram` prepares a 3-dimensional rendering of a color image histogram.
The pixels are drawn in the RGB space (not XY-plane) to show clustering and dispersion of RGB values.
This function can process pixels in other color systems (using EasyColor to convert), but the raw RGB image is required to display the pixels in their usual colors.

Magnification factors in all three directions (X = red, Y = green and Z = blue) can be given.
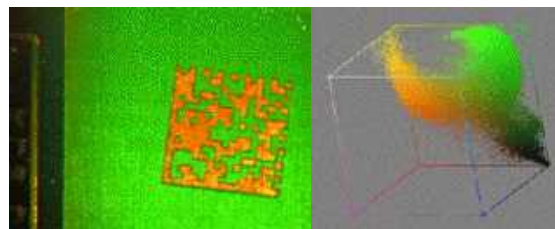
Color histogram rendering

2.7. Vector Types and Main Properties

A vector is a one-dimensional array of pixels (taken from an image profile or contour).

EVector is the base class for all vectors. It contains all non-type-specific methods, mainly for counting elements and serialization.
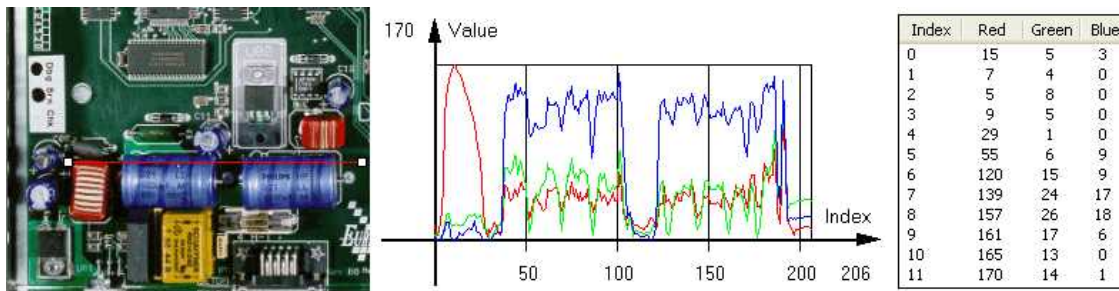
| Profile in a C24 image | RGB values plot along profile | RGB values array (EC24Vector) |

A vector manages an array of elements. Memory allocation is transparent, so vectors can be resized dynamically. Whenever a function uses a vector, the vector type, size and structure are automatically adjusted to suit the function needs.

The use of vectors is quite straightforward:

1. **Create a vector of the appropriate type**, using its constructor and pre-allocate elements if required.

Vector types

- EBW8Vector: a sequence of gray-level pixel values, often extracted from an image profile (used by EasyImage::Lut, EasyImage::SetupEqualize, EasyImage::ImageToLineSegment, EasyImage::LineSegmentToImage, EasyImage::ProfileDerivative, ...).
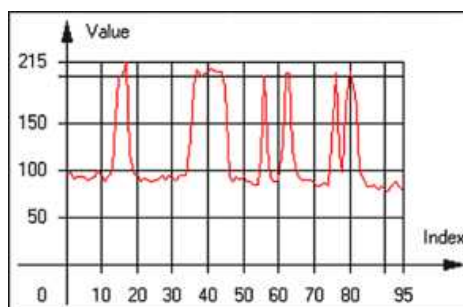
Graphical representation of an EBW8Vector **(see** Draw **method)**

- EBW16Vector: a sequence of gray-level pixel values, using an extended range (16 bits), mainly for intermediate computations.
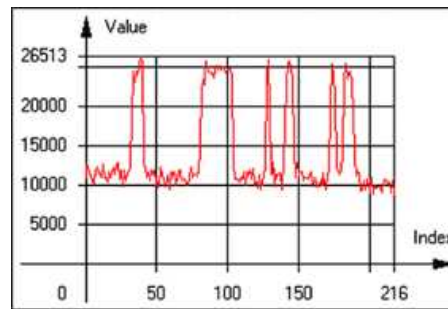
Graphical representation of an `EBW16Vector`

- `EBW32Vector`: a sequence of gray-level pixel values, using an extended range (32 bits), mainly for intermediate computations
  (used in `EasyImage::ProjectOnARow`, `EasyImage::ProjectOnAColumn`, ...).
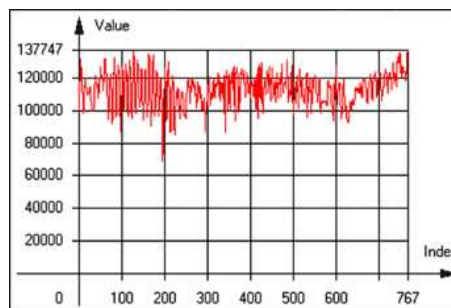
Graphical representation of an `EBW32Vector`

- `EC24Vector`: a sequence of color pixel values, often extracted from an image profile
  (used by `EasyImage::ImageToLineSegment`, `EasyImage::LineSegmentToImage`,
  `EasyImage::ProfileDerivative`, ...).
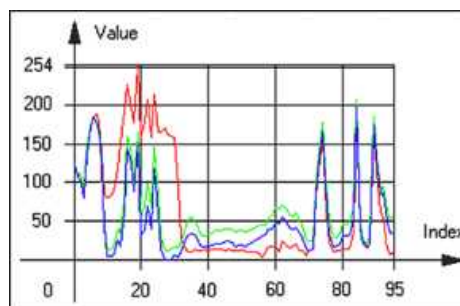
Graphical representation of an `EC24Vector`

- `EBW8PathVector`: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
  (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, ...).

Graphical representation of an `EBW8PathVector` **(see** `Draw` **method)**

- `EBW16PathVector`: a sequence of gray-level pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
  (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, …).
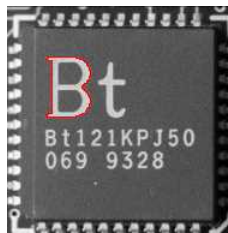
Graphical representation of an `EBW16PathVector` **(see** `Draw` **method)**

- `EC24PathVector`: a sequence of color pixel values, extracted from an image profile or contour, with corresponding pixel coordinates
  (used by `EasyImage::ImageToPath`, `EasyImage::PathToImage`, …).
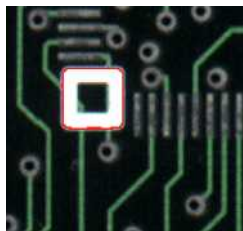
Graphical representation of an `EC24PathVector` **(see** `Draw` **method)**

- `EBWHistogramVector`: a sequence of frequency counts of pixels in a BW8 or BW16 image
  (used by `EasyImage::IsodataThreshold`, `EasyImage::Histogram`, `EasyImage::AnalyseHistogram`, `EasyImage::SetupEqualize`, …).
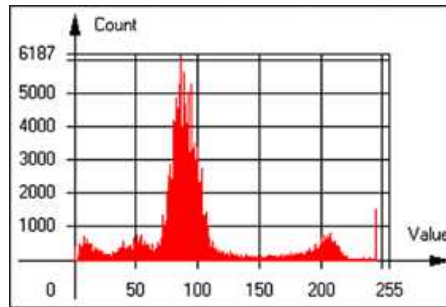
Graphical representation of an `EBWHistogramVector` **(see** `Draw` **method)**

- `EPathVector`: a sequence of pixel coordinates. The corresponding pixels need not be contiguous
  (used by `EasyImage::PathToImage` and `EasyImage::Contour`).
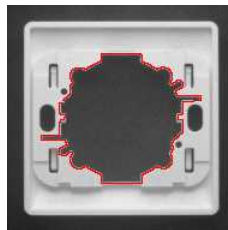
Graphical representation of an `EPathVector` **(see** `Draw` **method)**

- `EPeakVector`: peaks found in an image profile
  (used by `EasyImage::GetProfilePeaks`).
- `EColorVector`: a description of colors
  (used by `EasyColor::ClassAverages` and `EasyColor::ClassVariances`).

2. **Fill a vector with values**. First empty it, using the `EVector::Empty` member, then add elements one at a time by calling the `EC24Vector::AddElement` member. You can access any element by means of indexing.

3. **Access a vector element**, either for reading or writing. Use the brackets operator, for instance, `EC24Vector::operator[]`.

4. **Determine the current number of elements**, use member `EVector::NumElements`.

5. **Draw the vector**.
   A pixel vector is a plot of the element values as a function of the element index, so its graphical appearance depends on its type. You can draw a vector in a window. For legibility, the drawing should appear on a neutral background.
   Drawing is done in the device context associated to the desired window. By default, curves are drawn in blue, annotations are drawn in black. The following parameters can be defined: graphicContext, width, height, origin, origin, color0, color1, color2.
   The `EC24Vector` has three curves drawn instead of one, each corresponding to a color component. By default, red, blue and green pens are used.

2.8. ROI Main Properties

ROIs are defined by a width, a height, and **origin x and y coordinates**.
The origins are specified with respect to the top left corner in the parent image or ROI.
The ROI must be wholly contained in its parent image.
The processing/analysis time of a BW1 ROI is faster if `OrgX` and `Width` are multiples of 8.

Save and load

You can save or load an ROI as a separate image, to be used as if it was a full image. The ROIs perform **no memory allocation** at all and never duplicate parts of their parent image, the parent image provides them with access to its image data.

The image size of the new file must match the size of the ROI being loaded into it. The image around the ROI remains unchanged.

ROI Classes

An Open eVision ROI inherits parameters from the abstract class `EBaseROI`.

There are several ROI types, according to their pixel type. They have the same characteristics as the corresponding image types.

- `EROIBW1`
- `EROIBW8`
- `EROIBW16`
- `EROIBW32`
- `EROIC15`
- `EROIC16`
- `EROIC24`
- `EROIC24A`

Attachment

An ROI must be `attached` to a parent (image/ROI) with parameters that set the parent, position and size, and these links are updated transparently, avoiding dangling pointers.
A normal image cannot be attached to another image or ROI.

Nesting

Set and Get functions change or query the width, height and position of the origin of an ROI, with respect to its immediate or topmost parent image.

An image may accommodate an arbitrary number of ROIs, which can be nested in a hierarchical way. Moving the ROI also moves the embedded ROIs accordingly. The image/ROI classes provide several methods to traverse the hierarchy of ROIs associated with an image.
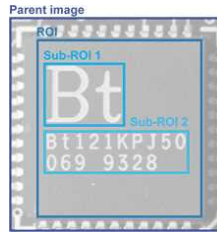
Nested ROIs: Two sub-ROIs attached to an ROI, itself attached to the parent image

Cropping

`CropToImage` crops an ROI which is partially out of its image. The resized ROI never grows. An exception is thrown if a function attempts to use an ROI that has limits that extend outside of the parents.

> **Note:** *(In Open eVision 1.0.1 and earlier, an ROI was silently resized or repositioned when placed out of its image and sometimes grew. If ROI limits extended outside parents, they were silently resized to remain within parent limits.)*

Resizing and moving

- ROIs can easily be resized and positioned by two functions and dragging handles:
  - `EBaseROI::Drag` adjusts the ROI coordinates while the cursor moves.
  - `EBaseROI::HitTest` informs if the cursor is placed over a dragging handle. Once the handle is known, the cursor shape can be changed by an `OnSetCursor` MFC event handler. HitTest is unpredictable if called while dragging is in progress.
    HitTest can be used in an OnSetCursor MFC event handler to change the cursor shape, or before a dragging operation like OnLButtonDown,
    (or EvSetCursor and EvLButtonDown in Borland/CodeGear's OWL)
    (or FormMouseMove and FormMouseDown in Borland/CodeGear's VCL).
    In VB6, MouseDown, MouseMove,MouseUp events return the current cursor position in twips rather than pixels, so conversion is mandatory.

2.9. Flexible Masks

ROIs vs flexible masks

ROIs and masks restrict processing to part of an image:

- "ROI Main Properties" on the previous page apply to all Open eVision functions. Using Regions of Interest accelerates processing by reducing the number of pixels. Open eVision supports hierarchically nested rectangular ROIs.
- Flexible Masks are recommended to process disconnected ROIs or non-rectangular shapes. They are supported by some EasyObject and EasyImage library functions.

Flexible Masks

A flexible mask is a BW8 image with the same height and width as the source image. It contains shapes of areas that must be processed and ignored areas (that will not be considered during processing):

- All pixels of the flexible mask having a value of 0 define the ignored areas.
- All pixels of the flexible mask having any other value than 0 define the areas to be processed.
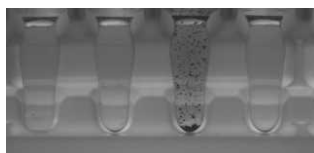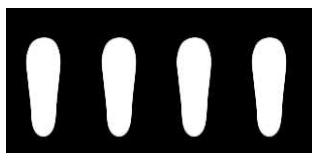
| **Source image** | **Associated mask** | **Processed masked image** |
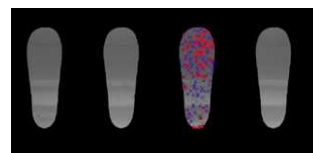
A flexible mask can be generated by any application that outputs BW8 images and by some EasyObject and EasyImage functions.

Flexible Masks in EasyImage
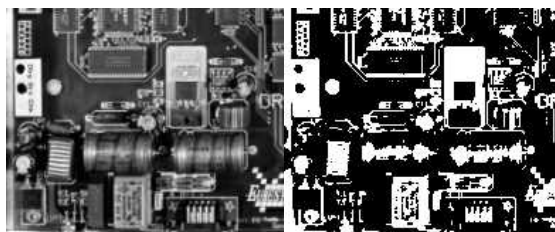
Source image (left) and mask variable (right)

Simple steps to use flexible masks in Easyimage

1. **Call the functions from EasyImage that take an input mask as an argument**. For instance, one can evaluate the average value of the pixels in the white layer and after in the black layer.

2. **Display the results.**
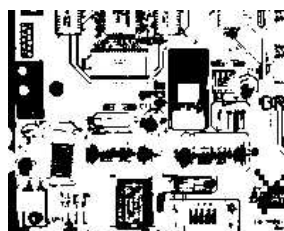
Resulting image

EasyImage Functions that support flexible masks

- `EImageEncoder::Encode` has a flexible mask argument for BW1, BW8, BW16, and C24 source images.

- AutoThreshold.

- Histogram (function `HistogramThreshold` has no overload with mask argument).

- RmsNoise, SignalNoiseRatio.

- Overlay (no overload with mask argument for BW8 source images).

- ProjectOnAColumn, ProjectOnARow (Vector projection).

- ImageToLineSegment, ImageToPath (Vector profile).

Flexible Masks in EasyObject

A flexible mask can be generated by any application that outputs BW8 images or uses the Open eVision image processing functions.
EasyObject can use flexible masks to restrict blob analysis to complex or disconnected shaped regions of the image.

If an object of interest has the same gray level as other regions of the image, you can define "keep" and "ignore" areas using flexible masks and `Encode` functions.

A flexible mask is a BW8 image with the same height and width as the source image.

- A pixel value of 0 in the flexible mask masks the corresponding source image pixel so it doesn't appear in the encoded image.

- Any other pixel value in the flexible mask causes the pixel to be encoded.

EasyObject functions that create flexible masks
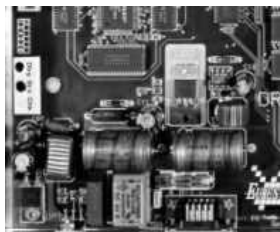
Source image

1) ECodedImage2::RenderMask: from a layer of an encoded image

1. To encode and extract a flexible mask, first construct a coded image from the source image.

2. Choose a segmentation method (for the image above the default method `GrayscaleSingleThreshold` is suitable).

3. Select the layer(s) of the coded image that should be encoded (i.e. white and black layers using minimum residue thresholding).

4. Make the mask image the desired size using `mask.SetSize(sourceImage.GetWidth(), sourceImage.GetHeight())`.

5. Exploit the flexible mask as an argument to `ECodedImage2::RenderMask`.
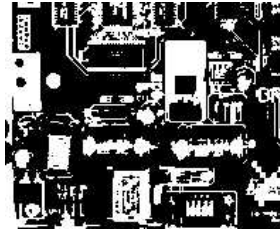
BW8 resulting image that can be used as a flexible mask

2) ECodedElement::RenderMask: from a blob or hole

1. Select the coded elements of interest.
2. Create a loop extracting a mask from selected coded elements of the coded image using `ECodedElement::RenderMask`.
3. Optionally, compute the feature value over each of these selected coded elements.
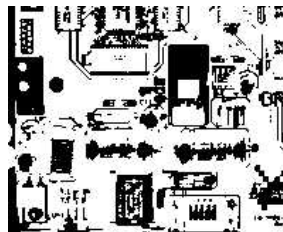
BW8 resulting image that can be used as a flexible mask

3) EObjectSelection::RenderMask: from a selection of blobs

`EObjectSelection::RenderMask` can, for example, discard small objects resulting from noise.
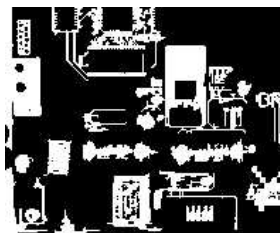
BW8 resulting image that can be used as a flexible mask

Example: Restrict the areas encoded by EasyObject
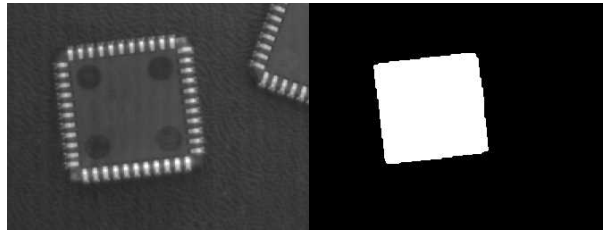
Find four circles (left) Flexible mask can isolate the central chip (right)

1. Declare a new `ECodedImage2` object.
2. Setup variables: first declare source image and flexible mask, then load them.
3. Declare an `EImageEncoder` object and, if applicable, select the appropriate segmenter. Setup the segmenter and choose the appropriate layer(s) to encode.
4. Encode the source image. Encoding a layer with just the area in the flexible mask is then pretty straightforward.
   We see that the circles are correctly segmented in the black layer with the grayscale single threshold segmenter:
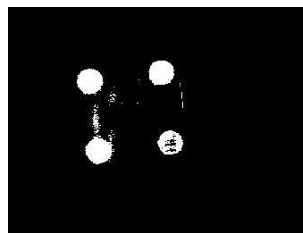
5. Select all objects of the coded image.
6. Select objects of interest by filtering out objects that are too small.
7. Display the blob feature by iterating over the selected objects to display the chosen feature.

2.10. Profile

Profile Sampling

A **profile** is a series of pixel values sampled along a line/path/contour in an image.

- `EasyImage::ImageToLineSegment` copies the pixel values along a given line segment (arbitrarily oriented and wholly contained within the image) to a vector. The vector length is adjusted automatically. This function supports flexible mask.

- A path is a series of pixel coordinates stored in a vector.
  `EasyImage::ImageToPath` copies the corresponding pixel values to the vector. This function supports flexible mask.

■ A **contour** is a closed or not (connected) path, forming the boundary of an object. `EasyImage::Contour` follows the contour of an object, and stores its constituent pixels values inside a profile vector.

Profile Analysis

The profile can be processed to find peaks or transitions:

■ A transition corresponds to an object edge (black to white or white to black). It can be detected by taking the first **derivative** of the signal (which transforms transitions (edges) into peaks) and looking for peaks in it.
`EasyImage::ProfileDerivative` computes the first derivative of a profile extracted from a gray-level image.
The `EBW8` data type only handles unsigned values, so the derivative is shifted up by 128. Values under [above] 128 correspond to negative [positive] derivative (decreasing [increasing] slope).

■ A peak is the portion of the signal that is above [or below] a given threshold - the maximum or minimum of the signal. This may correspond to the crossing of a white or black line or thin feature. It is defined by its:

■ Amplitude: difference between the threshold value and the max [or min] signal value.

■ Area: surface between the signal curve and the horizontal line at the given threshold.

`EasyImage::GetProfilePeaks` detects max and min peaks in a gray-level profile. To eliminate false peaks due to noise, two selection criteria are used. The result is stored in a peaks vector.

Profile Insertion Into an Image

`EasyImage::LineSegmentToImage` copies the pixel values from a vector or constant to the pixels of a given line segment (arbitrarily oriented and wholly contained within the image).

`EasyImage::PathToImage` copies the pixel values from a vector or a constant to the pixels of a given path.

3. Code Identification Tools

3.1. EasyBarCode - Reading Bar Codes

Reading Bar Codes

Bar code (EAN 13 symbology)

EasyBarCode can locate and read bar codes automatically.
Location can be performed manually for prototyping or when automatic mode results are unsatisfactory.

Workflow
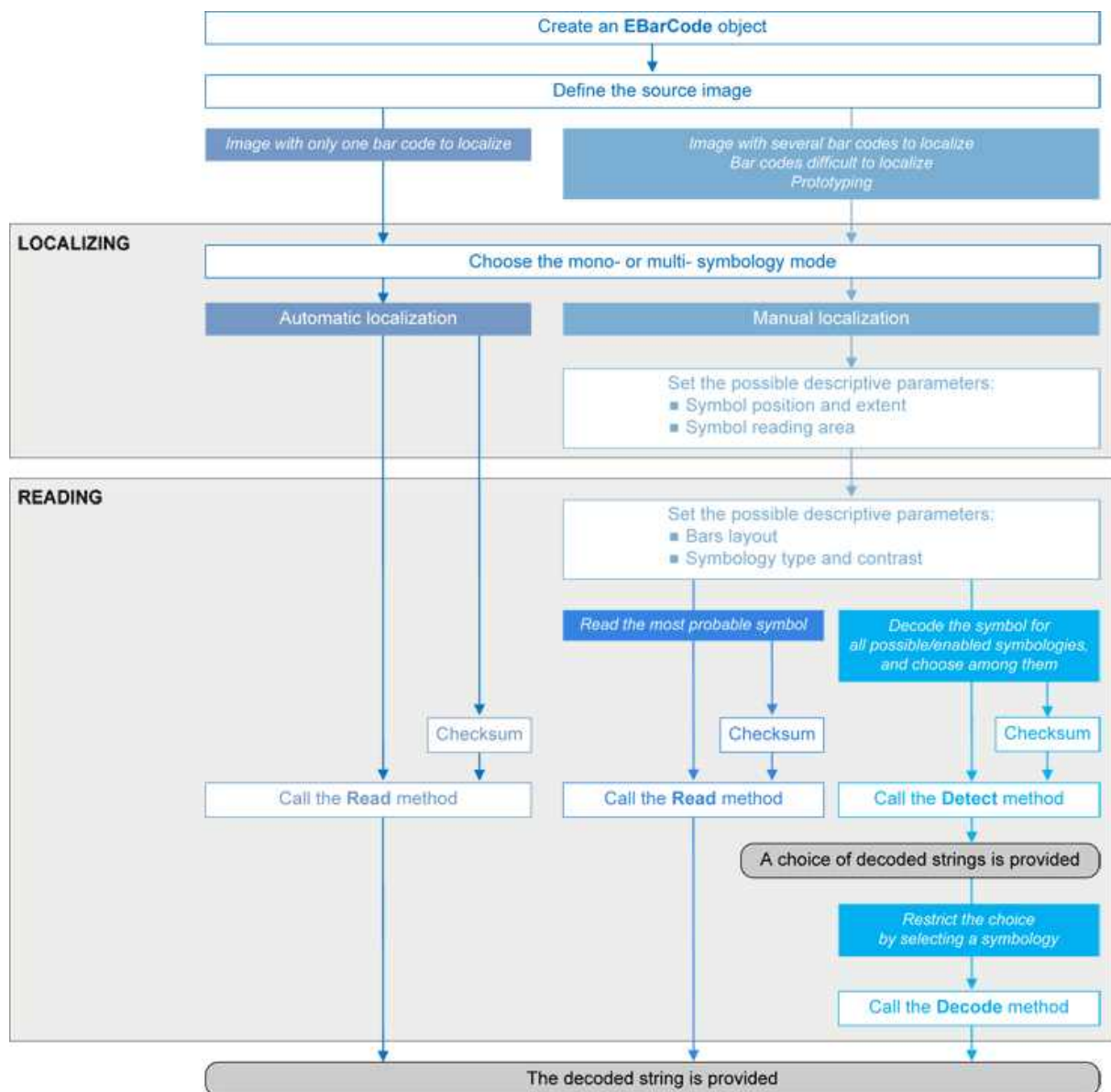
Bar code definition

A bar code is a 2D pattern of parallel bars and spaces of varying thickness that represents a character string. It is arranged according to an encoding convention (**symbology**) that specifies the character set and encoding rules.

- The bar code may be black ink on white background or inverted (white ink on black background).
- The bar code should be preceded and followed by a quiet zone of at least ten times the module width (smallest bar or space thickness).
- Bars should be surrounded below and above by a quiet zone of a few pixels.
- Bar and space widths must be greater than or equal to 2 pixels.

symbologies

A symbology defines the way a bar code is encoded.

Symbologies can be enabled in `StandardSymbologies` or `AdditionalSymbologies` parameters.

The standard symbologies are enabled by default:

- Code 39
- Code 128
- Code 2/5 5 Interleaved
- Codabar
- EAN 13*
- EAN 128
- MSI
- UPC A*
- UPC E

> **Note:** *EAN 13 and UPC A only differ by the layout of surrounding digits.*

Additional symbologies that are supported:

- ADS Anker
- Binary code
- Code 11
- Code 13
- Code 32
- Code 39 Extended (a super-set of Code 39)
- Code 39 Reduced (a subset of Code 39)
- Code 93
- Code 93 Extended
- Code 412 SEMI
- Code 2/5 3 Bars Datalogic
- Code 2/5 3 Bars Matrix
- Code 2/5 5 Bars IATA
- Code 2/5 5 Bars Industry
- Code 2/5 5 Compressed
- Code 2/5 5 Inverted
- Code BCD Matrix
- Code C.I.P
- Code STK
- EAN 8
- IBM Delta Distance A

- Plessey
- Telepen

Checksum

A checksum character enables the reader to check the barcode validity depending on the symbology:

- The checksum may be mandatory and must be checked by the reader.
- The checksum may be mandatory but may not need to be checked.
- The checksum and its verification may both be optional.

`VerifyChecksum` enables or disables (default) checksum verification.
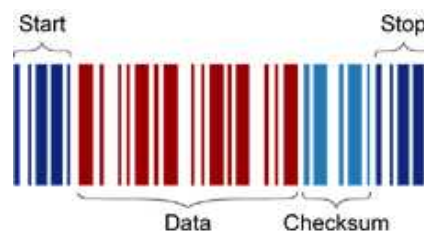
Bar code structure (Code 39)

Read a bar code

The **Automatic** mode reading algorithm locates a bar code in the field of view and Reads it.
If several bar codes are present, only one is located, like a straightforward hand-held bar code reader.

Before reading, the decoding symbologies must be specified in the StandardSymbologies, or AdditionalSymbologies properties.

Mono-symbology mode reads the bar code using the expected symbology type(s) and reports the encoded information (if readable) or the reason for failure (if not readable). There is only one interpretation for the character string.

Decoded bar code

Note: When the bar code contains \0x00 characters, the `std::string::c_str` method should not be used (since C-strings are terminated by the \0x00 character). An iterator over the characters should be used instead of a C-string.

Advanced features

Locate and Read bar code manually

If automatic localization fails or for prototyping purposes, the user can provide the **bar code position** and **reading area** to manually locate the code.

- **Bar code position** can be provided graphically by a bounding box around the bar code or by its parameters. If several symbols appear in the image, they can be processed one after the other.

- The **reading area** of the bar code is the area that is read. It should be wider than the bar code bounding box width, and less high than the bar code bounding box height. It may also be rotated relatively to the bar code bounding box, to take into account slanting bars (Advanced mode!).

EAN 128
(With Application Identifiers)

EAN 128
(With Application Identifiers)
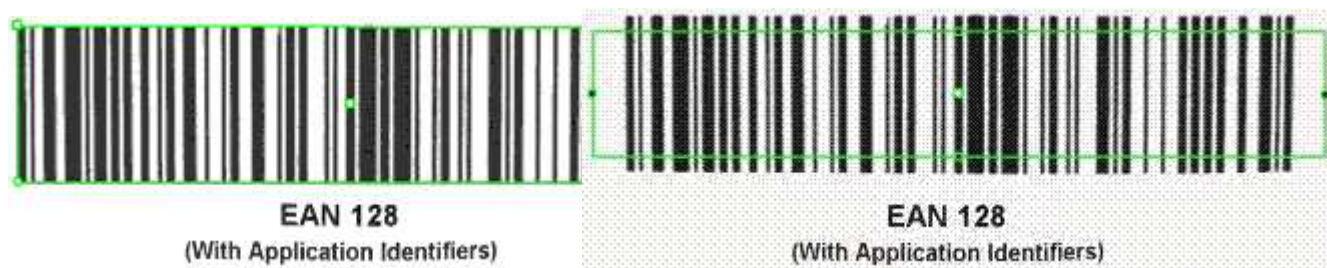
Bounding box — graphical appearance (manual location)

Reading area — graphical appearance (manual location)

Read all interpretations (multi-symbology mode)

Use `Detect` to report the number of possible symbologies in the `NumEnabledSymbologies` property, and list the data contents by decreasing likeliness.

Then call the `Decode` method in a loop, using `GetDecodedSymbology` to walk through the list of successful symbologies in decreasing order of likelihood.

Reading Mail Bar Codes
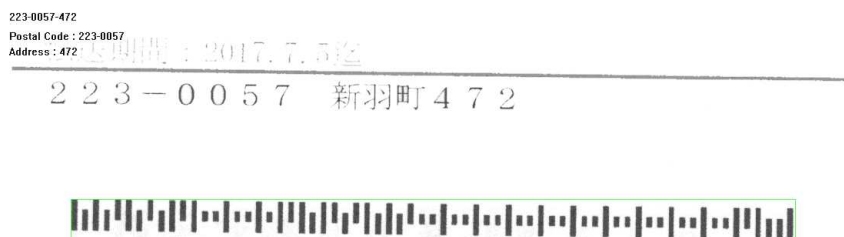
Mail bar code example

Specifications

The Mail Bar code Reader:

- Detects and decodes postal 4-state bar codes.

- Supports multiple mail bar codes in an image.

- Supports various symbologies.

- Supports the 4 main bar code orientations, with a tolerance of 3°.

- Detects bars that are at least 3 pixels wide.

Workflow

```
Create an EMailBarcodeReader Object
            │
            ▼
Select relevant symbologies
            │
            ▼
Call the Read method on the source image
            │
            ▼
A set of EMailBarcode objects is returned
            │
            ▼
Decoded contents are provided
```
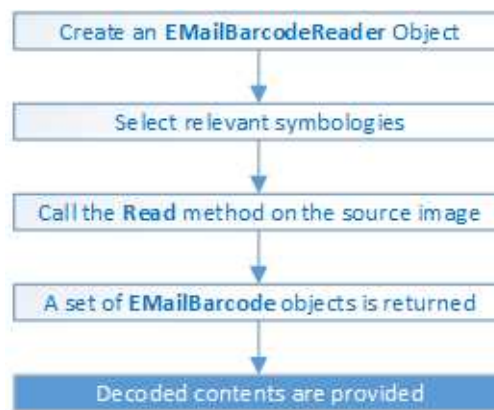
4-state bar codes

A 4-state bar code is a special kind of bar code where data is encoded on the height and position of the bars rather than their width.

Each bar can have one of 4 possible states:

☐ Short and centered

☐ Medium and elevated

☐ Medium and lowered

☐ Full height

Tracker
Ascender
Descender
Full Height

Mail bar code symbologies

The symbology of a mail bar code specifies how to decode the bar code and how to interpret its contents.

Every country uses its own flavor of mail bar code, or symbology. Some countries, like the US, even use multiple symbologies.

As of now, the Open eVision Mail Bar code Reader supports the following symbologies:

☐ US: PLANET, POSTNET and Intelligent Mail

□ Japan: Japan Post

Mail bar code orientation

The Open eVision Mail Bar code Reader is designed to be used in mail-handling machines. As such it is optimized to handle the 4 main orientations you encounter in such machines:

    □ No Rotation: The mail barcode is horizontal and read from left to right

    □ Rotated 90° to the right: The mail barcode is vertical and read from top to bottom

    □ Rotated 90° to the left: The mail barcode is vertical and read from bottom to top

    □ Rotated 180°: The mail barcode is upside down, horizontal, and read from right to left.

For each of these orientations, an additional rotation of -3 to 3 degrees is allowed.

Checksum

Some symbologies specify the presence of a checksum in the bar code data.

This checksum is an additional character computed from all others encoded characters. It enables the reader to check the decoded character string coherence.

- The Mail Bar code Reader allows the user to verify or not the checksum for all enabled symbologies.

- By default, checksum is not controlled.

- To enable or disable checksum verification for all enabled symbologies, set the `ValidateChecksum` property.

Reading the mail bar codes in an image

To read all the mail barcodes in a given image:

1. Create an `EMailBarcodeReader` object.

2. Optionally, select the relevant symbologies using the `ExpectedSymbologies` property.

   By default, Mail Bar code Reader will consider all supported symbologies.

3. Optionally, select the relevant orientations using the `ExpectedOrientations` property.

   By default, Mail Bar code Reader will test all supported orientations.

4. Call `Read` on the source image or ROI.

Each mail bar code detected is returned as an `EMailBarcode` object.

5. Each `EMailBarcode` objects contains the following information:

    □ The decoded string, using the `Text` property.

    □ The decoded string, split up in semantic parts, using the `ComponentStrings` property.

    □ The bar code orientation, using the `Orientation` property.

    □ The bar code position, using the `Position` property.

US Intelligent Mail bar code with highlighted position and decoded information

Advanced parameters

The advanced parameters of the `EMailBarcodeReader` object are:

- `EnableDottedBarcodes` activates the support for dotted barcodes (barcodes whose bars are printed with dots).

  By default, this property is set to false.
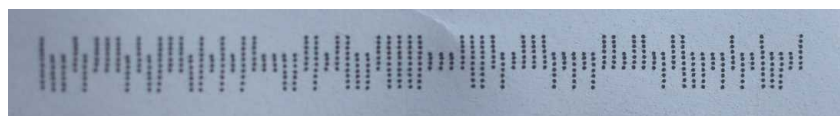
Dotted Mail Barcode

- `EnableClutteredBarcodes` activates the support for cluttered barcodes (barcodes in which some bars are connected).

  By default, this property is set to true.

Cluttered Mail Barcode

- `ValidateChecksum` activates the validation of the bar codes checksums, if present.

  By default, this property is set to false.

3.2. EasyMatrixCode - Reading Matrix Codes

EasyMatrixCode vs EasyMatrixCode2

Introducing EasyMatrixCode2

Important: *EasyMatrixCode2 is currently in beta release, so performance and features are not a complete image of the finished product.*

Starting with release 2.5, Open eVision introduces a new data matrix code reader functionality, named "EasyMatrixCode2". In the future, EasyMatrixCode2 should replace the current EasyMatrixCode (called "EasyMatrixCode1" from now on) with the following benefits:

- Ability to read multiple data matrix codes in an image.

- Improved consistency of reading and grading results.

- Improved consistency of processing time.

- Improved handling of deformed data matrix codes (this feature is not available yet and will be in a later release).

Note: *The relevant classes are stored in the EasyMatrixCode2 name space.*

Workflow

EasyMatrixCode1
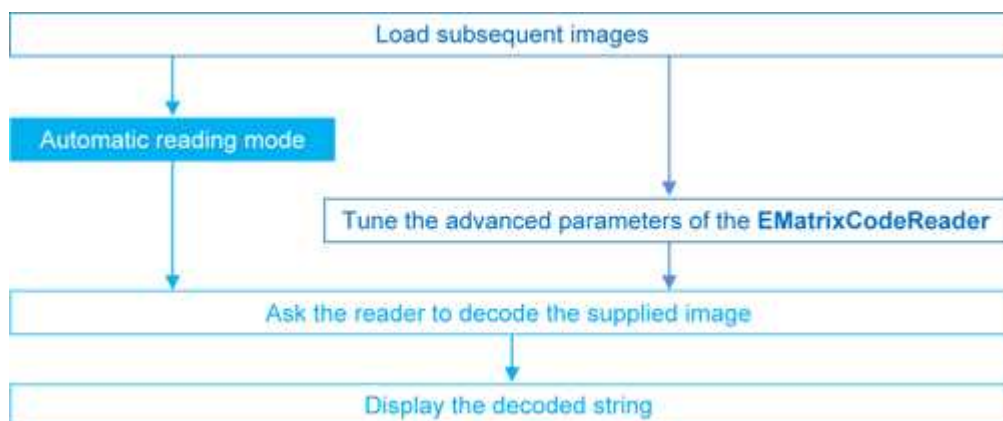
EasyMatrixCode2

1. Load the image.

2. Read the data matrix codes in the image using `EMatrixCodeReader.Read()`.

3. Loop on found data matrix codes.

4. Display the decoded text.

Specifications

Data matrix code definition

- A data matrix code is a two-dimensional rectangular array of black and white cells which conveys a string of characters (digits, letters and special characters).

  □ It is encoded to achieve maximum packing.

  □ Each cell corresponds to a bit of information.

  □ Additional redundant bits allow error correction for robust reading of degraded symbols.

- A data matrix code is located using the **Finder pattern**:

  □ The bottom and left edges of a Data Matrix code contain only black cells.

  □ The top and right edges have alternating cells.
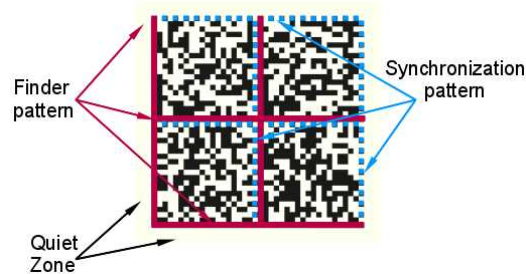
- A data matrix code is characterized by:

  □ Its logical size (number of cells).

  □ Its **encoding type**: ECC 000 (odd symbol sizes, deprecated) or ECC 200 (even symbol sizes)..

  Note: *The data matrix code definition is provided by ISO/IEC and approved as standard ISO/IEC 16022.*

Specifications
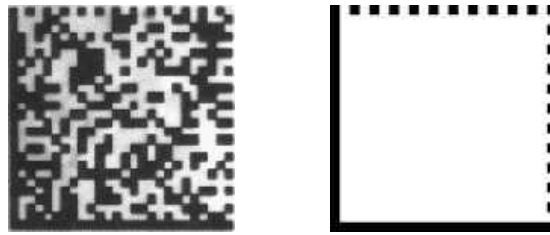
ECC 200, 26x26 cells data matrix code (left) and finder pattern (right)

In a single read operation, EasyMatrixCode locates, unscrambles, decodes, reads and grades the quality of grayscale 2D data matrix codes of any size, contrast, location and orientation (even viewed from the back on a transparent medium), providing they meet the following specifications:

☐ Minimum cell (= module) size: 3x3 pixels

☐ Maximum stretching (ratio between cell width and height): 2

☐ Minimum quiet zone (blank zone around the matrix code) width:
 - EasyMatrixCode1: 3 pixels
 - EasyMatrixCode2: 1 module

Reading a Matrix Code

Reading a matrix code

You can read matrix codes in an image automatically, using the `EMatrixCodeReader::Read ()` method. This method returns EMatrixCode instances, one per read data matrix code.

These instances include, among other things, for the found data matrix code:

☐ Its content (decoded string) ,

☐ Its position,

☐ Its logical size,

☐ Its encoding type,

☐ Methods to draw the data matrix code on the source image.

Note:
- EasyMatrixCode1 reads a single data matrix code in the image.
- EasyMatrixCode2 reads all the data matrix codes in the image at once.

Computing the print quality

To compute print quality indicators as defined by BC11, ISO 15415, ISO/IEC TR 29158 (formerly known as AIM DPM-1-2006) and SEMI T10-0701 standards, retrieve the grades with the `GetIso15415GradingParameters`, `GetIso29158GradingParameters` and `GetSemiT10GradingParameters` accessors of the EMatrixCode class.

> **Note:**
> *- With EasyMatrixCode1, enable the grading before using the Read method.*
> *- With EasyMatrixCode2, the grading is always enabled.*

Reading GS1 data matrix codes

EasyMatrixCode is able to find and decode GS1-compliant data matrix codes.

The GS1 standard adds semantic identifiers to the contents of a data matrix code, so that they are interpreted in an easy and consistent way.

GS1-compliant content is structured as follows:

```
]d2[GS1]{Id1}{Value1}[GS1]{Id2}{Value2}…
```

where:

- ☐ "]d2" is the string identifying a GS1-compliant stream,

- ☐ [GS1] is the GS1 escape character (0x1d),

- ☐ {Id} is an application identifier,

- ☐ {Value} is the value associated to that identifier.

Example

The string:

```
]d2[GS1]11180112[GS1]15190101
```

can be interpreted as follows:

- ☐ It contains two GS1 parts, 11180112 and 15190101.

- ☐ The first (11180112) is composed of the identifier 11 and the value 180112, meaning that the product has a production date (the meaning of identifier 11) of January 12th, 2018.

- ☐ The second (15190101) is composed of the identifier 15 and the value 190101, meaning that the product has a best before date (the meaning of identifier 15) of January 1st, 2019.

For more information, see https://www.gs1.org/

Learning a Matrix Code

> **Important:** *This topic applies to EasyMatrixCode1 only.*

To search for specific features and speed up your processing, learn a Matrix code model.

Workflow
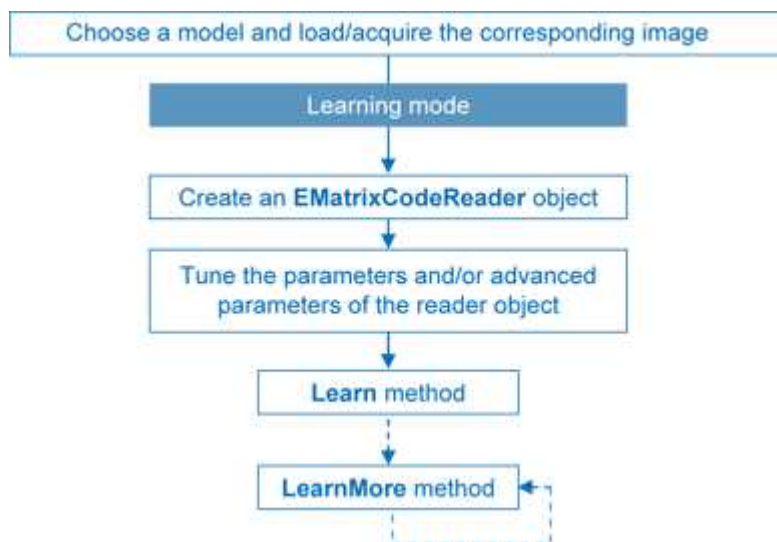
1. Load the image of the matrix code you want to learn.

2. Learn the model:

   □ Use the `Learn` method with `Contrast`, `Family`, `Flipping`, `Logical Size` parameters.

   □ If you need to learn several matrix codes, use `LearnMore` and pass additional sample images.

   □ Call `Learn` to replace `EMatrixCodeReader` parameters (calling `Learn` several times does not accumulate results, while `LearnMore` does).

3. Tune `search parameters` to be efficient and either:

   □ Read only matrix codes that match a sample matrix code,

   □ Or read only matrix codes that have the same properties (`Contrast`, `Family`, `Flipping`, `Logical Size`) as the learned one,

   □ Or disregard a search parameter of the learned matrix code `SetLearnMaskElement`, for example to read only unflipped matrixcodes. Just remove the default parameters, then add new ones.

4. Ask `EMatrixCodeReader` to decode the supplied image.

5. Display the `decoded string`.

6. Save the state of the reader object using `EMatrixCodeReader::Save`.

Restoring the state of an EMatrixCodeReader

To restore the state of an EMatrixCodeReader and use it to read a matrix code:

1. `Load` an image.

2. Restore the reader state from the given file using `EMatrixCodeReader::Load`.

3. `Read` the image.

4. Display the decoded string.

3.3. EasyQRCode - Reading QR Codes

EasyQRCode detects QR (Quick Response) codes in an image, decodes them, and returns their data.

Error detection and correction algorithms ensure that poorly-printed or distorted QR codes can still be read correctly.

Workflow
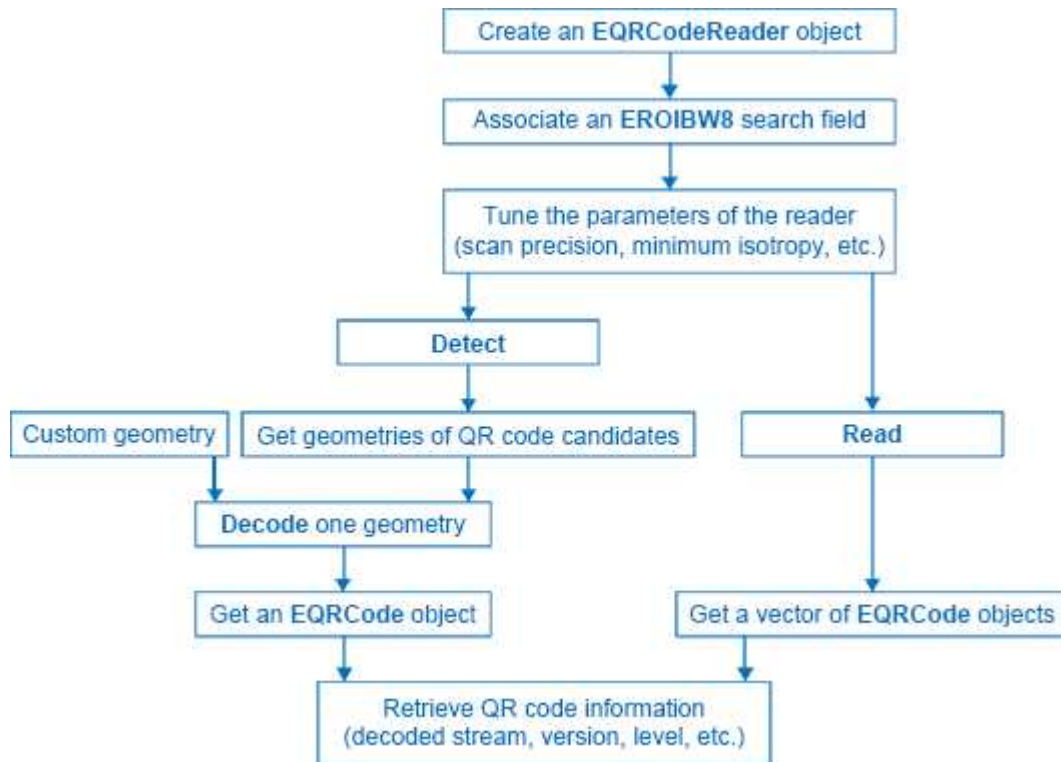
QR code definition

A QR code is a square array of dark and light dots. One dot (or "**module**") represents one bit of information.

QR codes contain various types of data and can be different models, versions, and levels. They always contain a message, metadata about alignment, size, format, and error correction bits. They comply with the international standard ISO/IEC 18004 (1, 2 and 2005).

QR code structure

The QR code symbol consists of an **encoding region**, containing data and error correction codewords, and of **function patterns**, containing symbol metadata and position data.

A QR code must be structured with the following elements:

- **Quiet zone**: blank margin around the QR code
- **Finder patterns**: recognizable zones identifying a QR code
- **Extension patterns**: markers for the alignment of the QR code (model 1)
- **Alignment patterns**: markers for the alignment of the QR code (models 2 and 2005)
- **Timing Patterns**: data giving the module size (in pixels)
- **Format information**: zones providing the QR code level

■ **Version information**: data giving the QR code size, for instance 25 x 25 modules (models 2 and 2005)

■ **Data contents and error correction codewords**: the primary information carried by the symbol, with additional information for error correction

Variants of this structure exist, according to the model, format, or version of the QR code. For instance, model 1 QR codes do not feature alignment patterns but extension patterns. Micro QR codes include only one finder pattern, and no alignment pattern. EasyQRCode can read all of them.
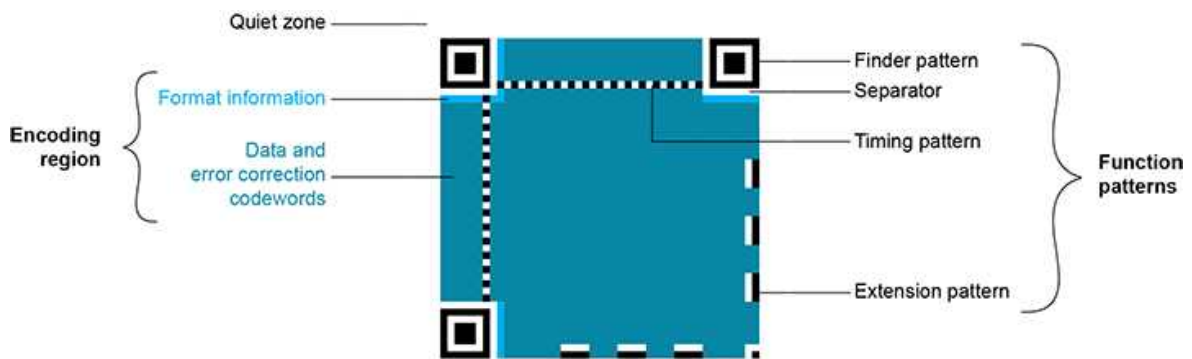
Structure of a model 1 QR code symbol
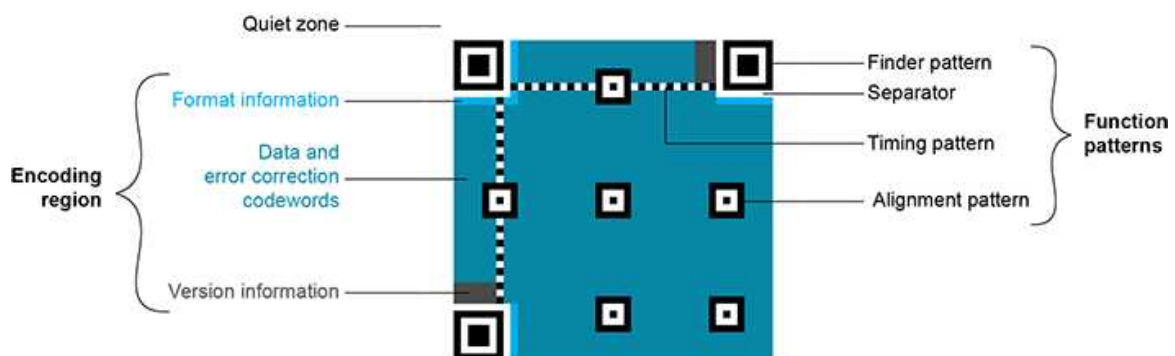
Structure of a QR code 2005 symbol
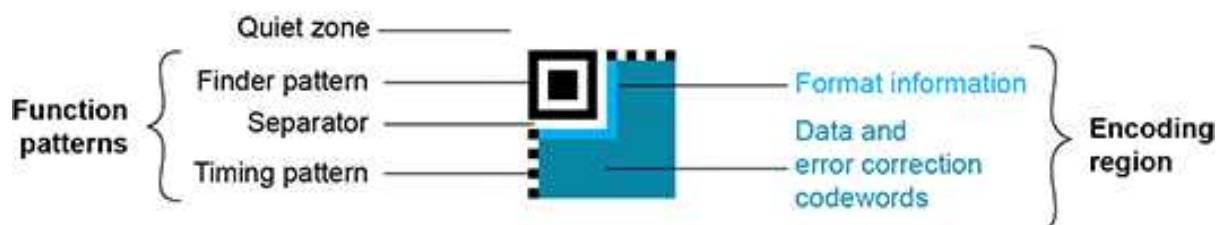
Structure of a Micro QR code symbol

Data types

The QR code data can be any mix of these types:

- Numeric data (0-9)
- Alphanumeric data (0-9, A-Z, /, $, %, etc.)
- Byte data
- Kanji characters

Models (Standards)

- **Model 1**: original QR code international standard, with versions ranging from 1 to 14. Note that the "version" of a QR code is the symbol size (in number of modules). It does not relate to the version of the standard, which is called the "model".
- **Model 2**: improvement of model 1. It provides versions from 1 to 40. It defines alignment patterns to improve reading of distorted QR codes, or QR codes printed on curved surfaces.
- **Model 2005**: improvement of model 2, including white-on-black QR codes, and mirror symbol orientation.
- **Micro QR codes**: (not yet supported) smaller QR codes, from version M1 to version M4. They have been introduced to save printing space.

Versions (Symbol Size)

- **QR codes**: from version **1** (21 x 21 modules) to version **40** (177 x 177 modules), with an increment of +4 x +4 modules (version 2: 25 x 25 modules, version 3: 29 x 29 modules, ..., version 39: 173 x 173 modules).
- **Micro QR codes**: (not yet supported) version **M1** (11 x 11 modules), version **M2** (13 x 13 modules), version **M3** (15 x 15 modules), version **M4** (17 x 17 modules).

Examples of QR codes
From left to right:
Micro QR code, version M3, 15 x 15 modules,
Model 2 QR code, version 4, 33 x 33 modules, 67-114 characters,
Model 2 QR code, version 40, 177 x 177 modules, 1852-4296 characters

Levels (Error Correction)

QR codes contain error correction data. The standard offers the following levels of error correction:

- **L**: (low) about 7% of codewords can be restored
- **M**: (medium) 15%
- **Q**: (quality) 25%
- **H**: (high) 30% (not available for Micro QR codes)

QR code geometry

When the QR code reader finds an array of dots that could match a QR code, it returns the "geometry" of this QR code candidate.

A QR code geometry is a set of points. It contains the coordinates of the corners of the QR code quadrilateral (bottom left, top left, top right, bottom right), and the coordinates of the finder pattern centers (bottom left, top left, top right).

QR code geometry

Read a QR code

Reading a QR code returns information about QR codes for which detection and decoding were successful.

This is equivalent to detecting and decoding all QR codes in the given search field (see advanced features).

Advanced features

Detect a QR code

1. Set a search field on an EROIBW8 image or tune the parameters to restrict the numbers of operations to process.

2. The QR code reader scans the image and searches for 3 finder patterns that could match a QR code, with the following requirements:

   - Minimum quiet zone (blank zone around the QR code) width: 3 pixels.
   - Minimum module size: 3 x 3 pixels.
   - Minimum isotropy: 0.5.
   - Maximum corner deformation: 15° (corner angles can range from 75° to 105°).

3. The reader returns QR code candidates, or the result of a detection, as a vector of geometries.

decode a QR code

1. The QR code reader decodes a QR candidate and returns the QR code: model, version, level, geometry and decoded stream of data.

   The decoded stream class consists of a coding mode (basic, FNC1/GS1, or FNC1/AIM), and an application indicator (if the coding mode is FNC1/AIM, otherwise 0). The decoded data can be accessed from each part of the decoded stream, according to its encoding (numeric, alphanumeric, byte, or Kanji). You can also get the raw bit stream (the bit data after unmasking and error correction, but before decoding as a vector of bytes).

2. The reader can report the amount of unused error correction.

   - Close to 1, very few errors were corrected when decoding the data. The decoding is highly reliable, and the QR code is of good quality.

   - Close to 0, many errors were corrected when decoding the data. The decoding is reliable, but the QR code quality is poor.

   - -1, error correction failed. Decoding was not performed.

Tuning parameters

Scan precision: You can change the scan precision to scan the search field with a fine (recommended for small QR codes), or coarse (recommended on medium to large QR codes) precision.

Minimum score: The QR code reader searches for this QR code finder pattern:

A perfect match returns a pattern finder score of 1.
Less accurate matches return lower scores.
The minimum score allowed by default is 0.65 - you can tune this.

Minimum isotropy: The isotropy of a QR code represents its rectangular deformation.
Perfectly square QR codes have an isotropy of 1 (short side divided by long side, whether the rectangle is vertical or horizontal).
EasyQRCode can detect rectangle QR codes with an isotropy down to 0.5. The default minimum isotropy is 0.8, it can be tuned from 0 to 1.

Square and rectangular QR codes (isotropy = 1, 0.5, and 0.5 from left to right)

Model and version: The QR code reader searches for QR codes of all models, and all versions. You can shorten the process by specifying the QR code model(s) and a range of versions (from 1 to 40) to be searched for.

4. Using Open eVision Studio

4.1. Selecting your Programming Language

When you start Open eVision Studio for the first time, the following welcome screen is displayed:
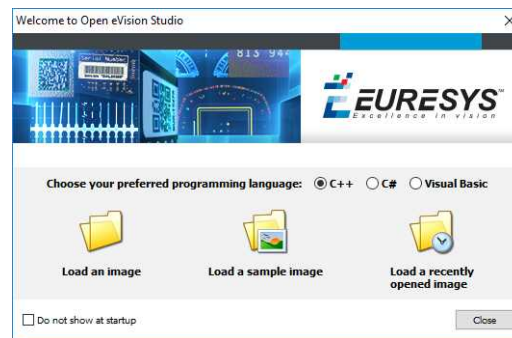
1. Select your programming language.

Your selection is saved and your programming language will be automatically selected next time you start Open eVision Studio.

> **Note:** *When you change your programming language, any script present in the scripting window is automatically deleted and the window content is reset.*

2. Click on one of the **Load** buttons to already load one or several images for later processing.

3. Check the **Do not show at startup** box to hide this welcome screen next time you start Open eVision Studio.

To access this welcome screen at any time, and change this setting, go to the Help > Welcome Screen menu.

4.2. Navigating the Interface
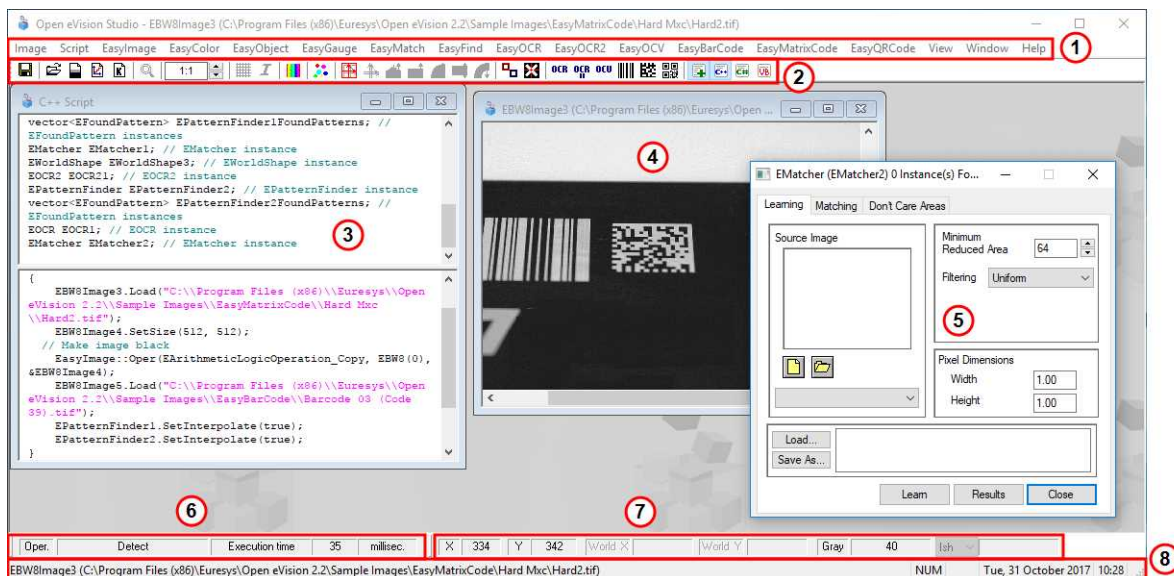
Open eVision Studio graphical user interface (GUI) is organized as follows:

1. The *main menu bar* gives you access to the functions and tools of all libraries.

Open eVision Studio does not require any license and allows you to test all libraries. Of course, if you copy code from Open eVision Studio in your own application but you do not have the required license, you will receive a "missing license" error at run-time.

2. The *main toolbar* gives you a quick access to main Open eVision objects such as images, shapes, gauges, bar codes, matrix codes...

3. The *script window* displays the code, in the programming language you selected, corresponding to the actions you perform in Open eVision Studio. You can save or copy this code in your own application at any time.

4. The *image windows* display the open images that you can process using the libraries and tools.

5. The *tool windows* enable you to easily configure all the available tools. The corresponding settings are automatically added in the script window for easy reuse.

Most tool windows are floating and you can easily move them outside the Open eVision Studio main window to make a better use of your screen size.

6. The *execution time bar* displays the precise time taken for the execution of the selected functions (measured in milliseconds or microseconds) on your computer. This accurate measurement helps you to evaluate the performance of your application.

7. The *color toolbar* displays current information such as the X and Y coordinates of the cursor on an image and the corresponding pixel value.

8. The *status bar* displays general information about the application such as the active image file path...

4.3. Running Tools on Images

Step 1: Selecting a Tool

Usually the first step, when using Open eVision Studio, is to select the library and the tool you want to use on your image.

To do so:

1. In the main menu bar, click on the library you want to use.

2. Click on the tool you want to use.

All libraries (except EasyImage, EasyColor and EasyGauge) expose only one tool named **New Xxx Tool**. *Some of these libraries also expose additional functions.*

3. In the dialog box, enter a **Variable name** for the variable that is automatically created and that will contain the result of the processing.
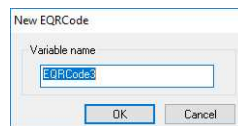
Example of variable creation dialog box for EasyQRCode

4. Click **OK**.

The selected tool dialog box opens.
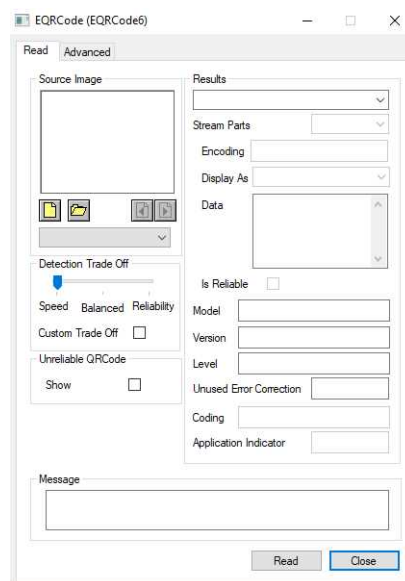
Example of variable creation dialog box for EasyQRCode

The next step is "Step 2: Opening an Image" below.

Step 2: Opening an Image

Once you have selected your library and your tool, you need to open an image to apply this tool.

In the **Source Image** area of the selected tool dialog box:

1. Open an image:

    □ Click on the ▣ **Open an Image** button and select one or several (using SHIFT and CTRL) images on your computer.

    □ Or select one of the images (or one of the ROIs, if any) already open in the drop-down list.

    Note: *You can select only images with an appropriate file format (JPG, PNG, TIFF or BMP) and in 8- and/or 24-bit depending on the library.*

2. If you selected several images, activate one with the ▣ **Load Previous** or ▣ **Load Next** buttons.

The tool is automatically applied on any loaded image and, at this stage, the result is displayed based on the tool default settings.

The next step is "Step 3: Managing ROIs" below.

Step 3: Managing ROIs

In some cases, most often to decrease the processing time or to single-out the object you want to read, you do not want to process the whole image but only one or several well defined rectangular parts of this image, or ROIs (Regions Of Interest).

In Open eVision, ROIs are attached to an image and exist only as long as the parent image is available.

Creating a ROI

1. Open the image:

    □ If the image is already open, activate the corresponding image window.

    □ If the image is not open yet, go to the main menu: **Image** > **Open…** to open one.

2. To create an ROI, go to the main menu: **Image** > **ROI Management….**

The **ROI Management** window is displayed as illustrated below.
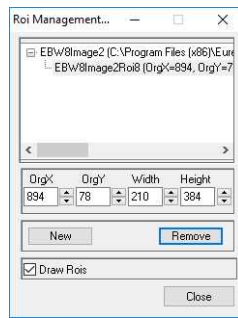
3. Select the image in the tree.

4. Click on the **New** button.

5. In the dialog box, enter a **Variable name** for the new ROI.

The ROI is represented as a color rectangle on your image as illustrated below.
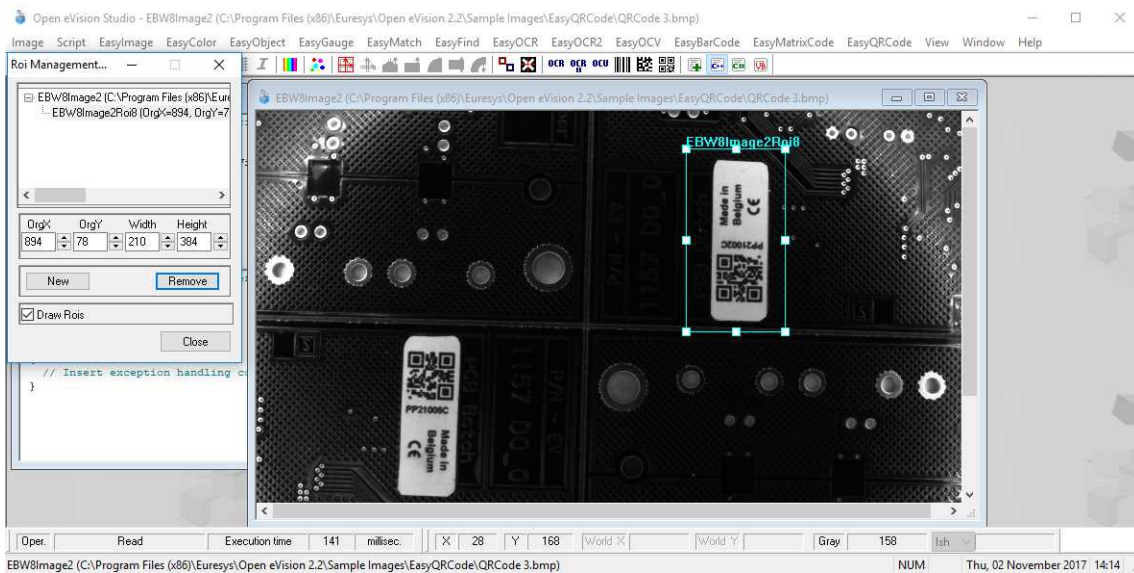
6. Drag the ROI corner and side handles to move it to the required position.

7. Click on the **Close** button to close the **ROI Management** window .

The next step is "Step 4: Configuring the Tool" on the next page.

Managing ROIs

You can add, change and remove ROIs.

An image can have several ROIs. Each ROI can be attached directly to the image (meaning that its position is relative to the image) or to another ROI (meaning that its position is relative to this 'parent' ROI).

1. To manage ROIs, go to the main menu: **Image** > **ROI Management...**.

The **ROI Management** window is displayed with the ROI relation tree as illustrated below.

If the **Draw Rois** box is checked, all ROIs are displayed on the image with a different color.
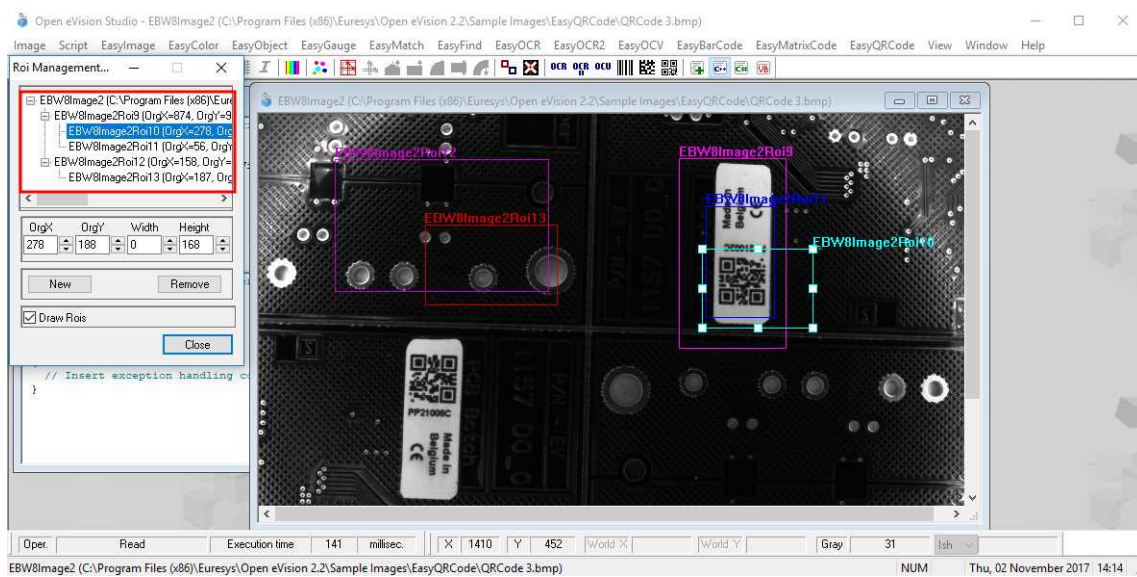
2. Select an ROI in the ROI relation tree.

3. Drag the ROI corner and side handles to change the position and size of the selected ROI (as well as the position of all ROIs attached to it if any).

4. Click on the **New** button to add a new ROI attached to the selected ROI.

Select the image at the top of the ROI relation tree to attach the ROI directly to the image.

5. Click on the **Remove** button to delete the selected ROI (and all ROIs attached to it if any).

6. Click on the **Close** button to close the **ROI Management** window.

Step 4: Configuring the Tool

Once your image, including its ROIs if you created some, is ready, you need to configure your tool.

In the tool window:

1. Open the various tabs.

When you create a new tool, all parameters are set with their default value.
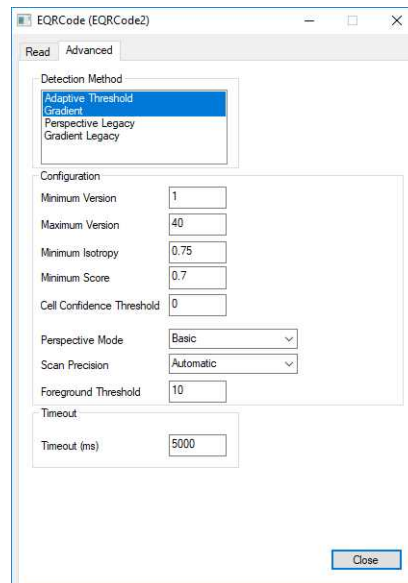
Example of the parameter tab of an EasyQRCode tool

2. In each tab, set the value of the parameters as desired.

Please refer to the "Functional Guide" and to the "Reference Manual" for detailed information about the parameters, their function and their default value.

For specific actions such as learning or using gauges, please refer to the "Functional Guide".

3. Run the tool and analyze the results as described in the next step "Step 5: Running the Tool and Checking Execution Time" below.

Step 5: Running the Tool and Checking Execution Time

Once your tool parameters are set, run your tool and, if desired, check the execution time on your computer.

In the tool window:

1. Click on the **Read**, **Detect**, **Results** or **Execute** button (depending on the library function), to run the tool on the selected image.

2. Check the results on the image and in the Results field or area as illustrated below.
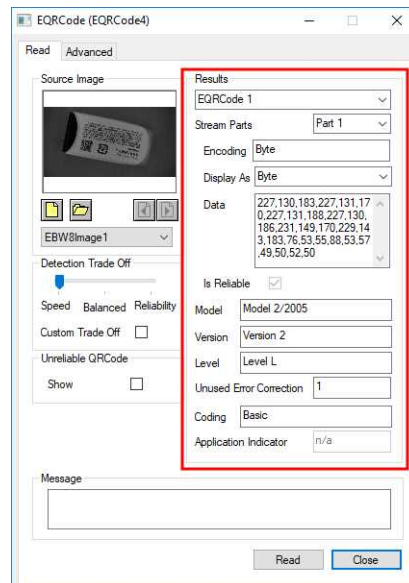
Example of results after reading a QRCode

3. If you do not have the expected results:

☐ Try to change your parameters (start with default values then change one parameter at a time).

☐ If you image is not good enough, try to enhance it as described in .

4. Check the execution time in the execution time bar at the bottom left of the main Open eVision Studio window.
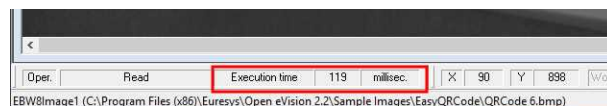
The execution time
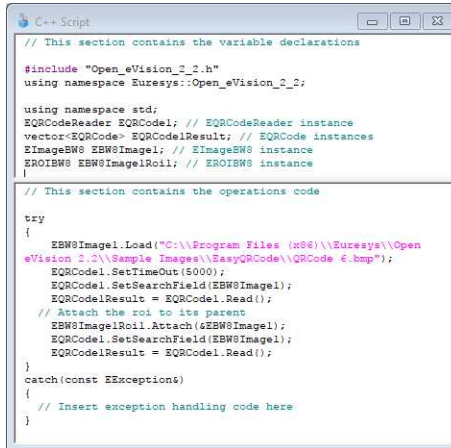
The execution time is the actual time that the processing took as measured on your computer. It depends your computer processor, memory, operating system... and, of course, on the processor load at the time of execution. Thus this execution time slightly varies from execution to execution.

5. To get a more representative execution time, click on the **Read**, **Detect**, **Results** or **Execute** button several times and calculate the mean execution time.

6. If your application requires that you reduce the execution time, try:

☐ To change the tool parameters,

☐ To add one or several ROIs on your image,

☐ To enhance your image.

The next step is "Step 6: Using the Generated Code" on the next page.

Step 6: Using the Generated Code

By default, Open eVision Studio translates all the operations you perform in the interface into code in the language you selected as illustrated below.

Once your tool results suit you, you can save or copy this generated code to use it in your own application.

Copy and paste the code in your application

In the script window:

1. Select the code section you want to copy.

2. Right click on this code and click **Copy** in the menu.

3. Go to you development environment tool and paste the code in place.

Save the code

1. Go to the **Script** menu.

2. Click on **Save Script As....**

3. Enter a file name and path to save the code as a text file.

Manage the generated code

In the **Script** menu, you can:

- ☐ Select the programming language (please note that if you change the language, the script window content is automatically deleted).

- ☐ Activate or deactivate the **Script Code Generation**. Deactivate this option if you want to perform some operations without saving them as code.

4.4. Pre-Processing and Saving Images

When should you pre-process your images?

Of course, the best situation is to set up your image acquisition system to have good and easy to process images so the Open eVision tools run smoothly and efficiently.

If this is not possible or easy to achieve, you can pre-process your images or your ROIs to enhance and prepare them for the Open eVision tool you want to run.

Using the various available functions, you can adjust the gain and offset of your image, apply a convolution, threshold, scale, rotate and white balance your image, enhance contours... using EasyImage and EasyColor functions.

Pre-processing images

The difference between pre-processing an image and running tools is that the pre-processing generates a new image while the tools mainly extract and retrieve information from the image without changing it.

To pre-process an image or an ROI:

1. In the main menu bar, click on the library you want to use (EasyImage or EasyColor).

2. Click on the function you want to use.

Most function dialog boxes are similar to the one illustrated below with 2 image selection areas and a parameter setting area.
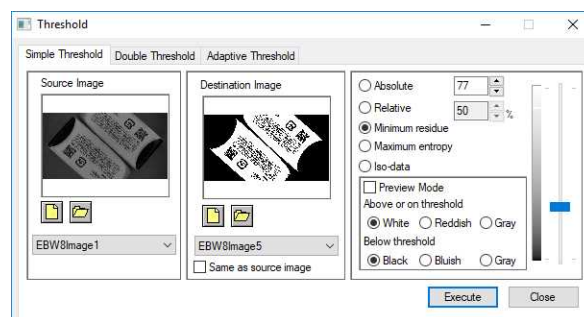
Example of a pre-processing dialog box (Threshold with EasyImage)

3. If there are multiple versions for your selected function, open the corresponding tab.

4. In the **Source Image** area, open the source image (as described in "Step 2: Opening an Image" on page 56).

5. In the **Destination Image** area, open or create a new destination image.

6. Set your parameters.

7. Click on the **Execute** button.

The pre-processed image is available in the destination image as illustrated below.
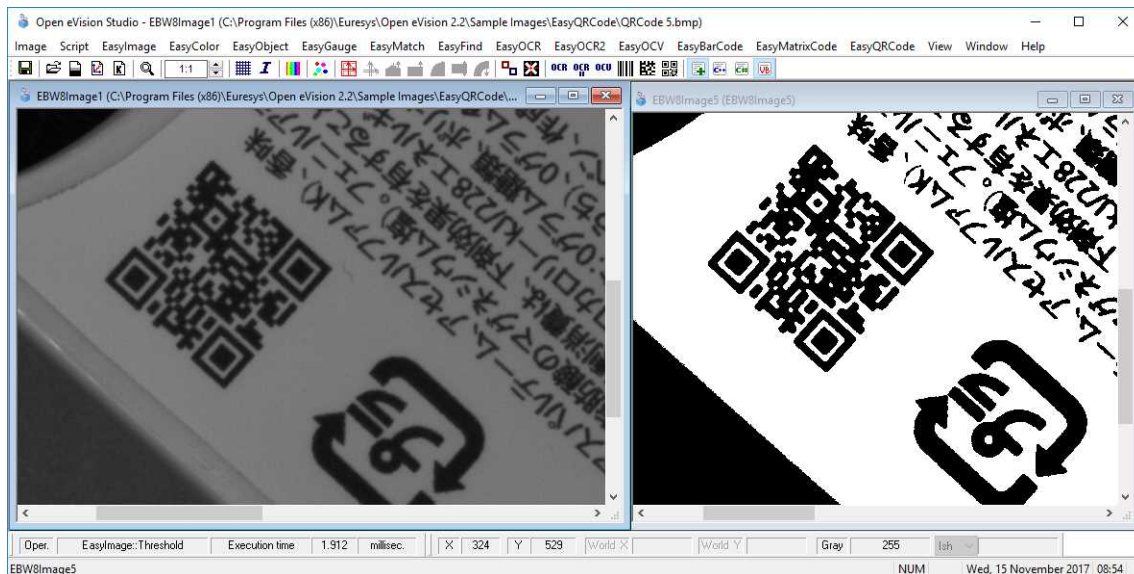
Source and destinations images (Threshold with EasyImage)

8. If you want to use the destination image outside of Open eVision Studio, save it as described below.

Saving an image

1. Click in the image you want to save to activate it.

2. To open the save menu either:

☐ Right-click in the image

☐ Or open the main menu > **Image**

3. Click on **Save as…**.

4. Select the file format (JPEG, JPEG2000, PNG, TIFF or Bitmap).

5. Enter a name and select a path.

6. Click on the **Save** button.

5. Tutorials

5.1. EasyBarCode

Reading Bar Codes Automatically

Objective

Following this tutorial, you will learn how to perform automatic reading of multiple bar codes.

You'll need first to load multiple source images (step 1). The reading is then automatically performed on each image (step 2).

Each bar code is automatically detected and decoded

Step 1: Load the source images

1. From the main menu, click **EasyBarCode**, then **New BarCode Tool**.

2. Keep the default variable name, and click **OK**.

3. In the **AutoRead** tab, click the **Open** icon of the Source Image area, and load the image files `EasyBarCode\Barcode 01.tif` to `Barcode 10.tif`. Use the shift key to select multiple files.

4. Keep the default variable name, and click **OK**. The last image appears.

Step 2: Read the bar codes automatically

1. The bar code is automatically detected and decoded. The graphic result appears on the image, while the data content and the corresponding symbology are displayed in the Decoded Symbology area. It is not necessary to click **Read** once a new image appears. However, clicking **Read** will insert the corresponding code into the script window.

2. In the **Results** tab, find more information about the bar code. As a bar code might have a meaning under different symbologies, all possible contents are listed by decreasing likeliness.

3. In the **AutoRead** tab, click the **Load Next** and **Load Previous** icons to browse through images 01 to 06.

The image files appear, and each bar code is automatically detected and decoded. The bar code properties are updated.

4. To decode the remaining bar codes, we have to enable the additional symbologies.

5. In the **Symbologies** tab, click the **Toggle All** button of the Additional area.

6. In the **AutoRead** tab, click the **Load Next** and **Load Previous** icons and browse the remaining images.

5.2. EasyMatrixCode

Reading Data Matrix Codes Automatically

"Automatic Reading" on page 76

Objective

Following this tutorial, you will learn how to use EasyMatrixCode to detect and decode automatically Data Matrix codes in multiple files.

You'll need first to load multiple source images (step 1). The reading is then automatically performed on each image (step 2). You can also grade the printing quality of each matrix code (step 3).
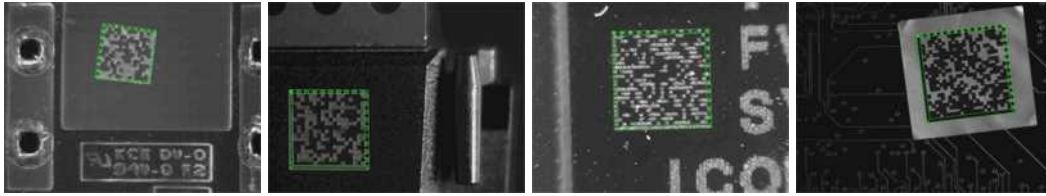
Each Data Matrix code is automatically detected and decoded

Step 1: Load the source images

1. From the main menu, click **EasyMatrixCode**, then **New MatrixCode Tool**.

2. Keep the default variable name for the new matrix code reader, and click **OK**.

3. In the **Read** tab, click the **Open** icon of the Source Image area, and load the image files `EasyMatrixCode\AutoRead\AutoRead 01.tif` to `AutoRead 04.tif`. Use the shift key to select multiple files.

4. Keep the default variable name for the new image, and click OK. The last image appears.

Step 2: Read the Data Matrix codes automatically

1. The Data Matrix code is automatically detected and decoded. The matrix code reference corner is highlighted with a bold cross mark. It is not necessary to click **Read** once a new image appears. However, clicking **Read** will insert the corresponding code into the script window.

2. In the **Results** area, find more information about the matrix code, such as the decoded string.

3. In the **Read** tab, click the **Load Next** and **Load Previous** icons. The image files appear, and each Data Matrix code is automatically detected and decoded. The matrix code properties are updated. If no Data Matrix code could been located in the image, an error message is displayed in the Message field.

Step 3: Grade Data Matrix code printing quality

1. In the **Print Quality** tab, select the **Compute Grading** check-box.

2. Click **Apply**.

For each printing quality parameter, the corresponding value and its grade equivalent appear. An A grade means a good quality, while an F grade indicates a poor one.

Learning a Data Matrix Code and Creating an EasyMatrixCode Model File

"Reading with Prior Learning" on page 76

Objective

Following this tutorial, you will learn how to use EasyMatrixCode to learn a Data Matrix code, and save it as an EasyMatrixCode model file.

You'll need first to load a source image (step 1), and learn the matrix code (step 2). Then you'll save the learned matrix code as a new model file (step 3). You can also add new learned matrix codes to an existing model if needed (step 4).
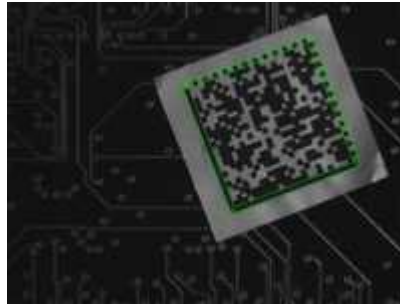
The Data Matrix code has been learned

Step 1: Load the source image

1. From the main menu, click **EasyMatrixCode**, then **New MatrixCode Tool**.

2. Keep the default variable name for the new matrix code reader object, and click **OK**.

3. In the **Learn** tab, click the **Open** icon of the Source Image area, and load the image file `EasyMatrixCode\Label\Label 4.tif`.

4. Keep the default variable name for the new image object, and click **OK**.

Step 2: Learn the Data Matrix code

● In the **Learn** tab, click **Learn**.

The Data Matrix code is detected and decoded without error.

The graphical result appears on the image.

The properties of the learned matrix code are updated in the dialog box.

Step 3: Save the model file

1. In the **Learn** tab, click the **Save As...** button.

2. Type a file name for the new EasyMatrixCode model file. Its extension will be `.mx2`.

3. Click **Save**.

Step 4: Learning more Data Matrix codes

1. In the **Read** tab, click the **Open** icon of the source image area, and load the image file `EasyMatrixCode\PCB Code\PCB Code 3.jpg`.

2. Keep the default variable name for the new image object, and click **OK**.

3. An error message is displayed in the Message area of the **Read** tab. The matrix code can not be read, since the reader uses the model from the "Label 4" image. You need to learn the "PCB Code 3" matrix code, and add it to the model.

4. In the **Learn** tab, click **Learn More**. The Data Matrix code is detected and decoded without error. The graphical result appears on the image. The properties of the learned matrix code are updated in the dialog box.

5. Using **Learn More** rather than Learn involves that the "Label 4" model is not replaced by the "PCB Code 3" model, but both are now included in the same model. In the Read tab, the "PCB Code 3" matrix code is correctly read. Select the "Label 4" image in the drop-down list of the source image area. The "Label 4" matrix code is still read without error, which means that both learned matrix codes have been kept.

6. Finally, save the model again (refer to step 3). The new matrix code has been added.

6. Code Snippets

6.1. Basic Types

Loading and Saving Images

```
/////////////////////////////////////////////////////////
// This code snippet shows how to load and save an image. //
/////////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcImage;
EImageBW8 dstImage;

// Load an image file
srcImage.Load("mySourceImage.bmp");

// ...

// Save the destination image into a file
dstImage.Save("myDestImage.bmp");

// Save the destination image into a jpeg file
// The default compression quality is 75
dstImage.Save("myDestImage.jpg");

// Save the destination image into a jpeg file
// set the compression quality to 50
dstImage.SaveJpeg("myDestImage50.jpg", 50);
```

Interfacing Third-Party Images

```
/////////////////////////////////////////////////////////
// This code snippet shows how to link an Open eVision image //
// to an externally allocated buffer.                        //
/////////////////////////////////////////////////////////

// Images constructor
EImageBW8 srcImage;

// Size of the third-party image
int sizeX;
int sizeY;

//Pointer to the third-party image buffer
EBW8* imgPtr;

// ...

// Link the Open eVision image to the third-party image
// Assuming the corresponding  buffer is aligned on 4 bytes
srcImage.SetImagePtr(sizeX, sizeY, imgPtr);
```

Retrieving Pixel Values

```
/////////////////////////////////////////////////////////
```

```
// This code snippet shows the recommended method (fastest)  //
// to access the pixel values in a BW8 image                 //
/////////////////////////////////////////////////////////////
```

```
EImageBW8 img;
```

```
OEV_UINT8* pixelPtr;
OEV_UINT8* rowPtr;
OEV_UINT8  pixelValue;
OEV_UINT32 rowPitch;
OEV_UINT32 x, y;
```

```
rowPtr = reinterpret_cast <OEV_UINT8*>(img.GetImagePtr());
rowPitch = img.GetRowPitch();
```

```
for (y = 0; y < height; y++)
{
    pixelPtr = rowPtr;

    for (x = 0; x < width; x++)
    {
        pixelValue = *pixelPtr;

        // Add your pixel computation code here

        *pixelPtr = pixelValue;
        pixelPtr++;
    }

    rowPtr += rowPitch;
}
```

ROI Placement

```
/////////////////////////////////////////////////////////////
// This code snippet shows how to attach an ROI to an image  //
// and set its placement.                                    //
/////////////////////////////////////////////////////////////
```

```
// Image constructor
EImageBW8 parentImage;
```

```
// ROI constructor
EROIBW8 myROI;
```

```
// ...
```

```
// Attach the ROI to the image
myROI.Attach(&parentImage);
```

```
//Set the ROI position
myROI.SetPlacement(50, 50, 200, 100);
```

Vector Management

```
/////////////////////////////////////////////////////////////
// This code snippet shows how to create a vector, fill it   //
// and retrieve the value of a given element.                //
/////////////////////////////////////////////////////////////
```

```
// EBW8Vector constructor
EBW8Vector ramp;
```

```
// Clear the vector
ramp.Empty();
```

```
// Fill the vector with increasing values
for(int i= 0; i < 128; i++)
{
        ramp.AddElement((EBW8)i);
}
```

```
// Retrieve the 10th element value
EBW8 value= ramp[9];
```

Exception Management

```
/////////////////////////////////////////
// This code snippet shows how to manage  //
// Open eVision exceptions.               //
/////////////////////////////////////////
```

```
try
{
        // Image constructor
        EImageC24 srcImage;
```

```
        // ...
```

```
        // Retrieve the pixel value at coordinates (56, 73)
        EC24 value= srcImage.GetPixel(56, 730);
}
```

```
catch(Euresys::Open_eVision_1_1::EException exc)
{
        // Retrieve the exception description
        std::string error = exc.What();
}
```

6.2. EasyBarCode

Reading a Bar Code

```
//////////////////////////////////////////////////
// This code snippet shows how to read a bar code  //
//////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Bar code reader constructor
EBarCode reader;

// String for the decoded bar code
std::string result;

// ...

// Read the source image
result = reader.Read(&srcImage);
```

Reading a Bar Code Following a Given Symbology

```
//////////////////////////////////////////////////////////////////
// This code snippet shows how to enable a given symbology,  //
// enable the checksum verification, perform the bar code    //
// detection and retrieve the decoded string.                //
//////////////////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Bar code reader constructor
EBarCode reader;

// String for the decoded bar code
std::string result;

// ...

// Disable all standard symbologies
reader.SetStandardSymbologies(0);

// Enable the Code32 symbology only
reader.SetAdditionalSymbologies(ESymbologies_Code32);

// Enable checksum verification
reader.SetVerifyChecksum(true);

// Detect all possible meanings of the bar code
reader.Detect(&srcImage);

// Retrieve the number of symbologies for
// which the decoding process was successful
int numDecoded = reader.GetNumDecodedSymbologies();
```

```
if (numDecoded > 0)
{
        // Decode the bar code according to the Code32 symbology
        result = reader.Decode(ESymbologies_Code32);
}
```

Reading a Bar Code of Known Location

```
//////////////////////////////////////////////////////
// This code snippet shows how to specify the bar code  //
// position and perform the bar code reading.           //
//////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Bar code reader constructor
EBarCode reader;

// String for the decoded bar code
std::string result;

// ...

// Disable automatic bar code detection
reader.SetKnownLocation(TRUE);

// Set the bar code position
reader.SetCenterXY(450.0f, 400.0f)
reader.SetSize(250.0f, 110.0f);
reader.SetReadingSize(1.15f, 0.5f);

// Read the bar code at the specified location
result = reader.Read(&srcImage);
```

Reading a Mail Bar Code

```
//////////////////////////////////////////////////////
// This code snippet shows how to read Mail Barcodes  //
// and retrieve the decoded data.                     //
//////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Mail barcode reader constructor
EMailBarcodeReader reader;

// Select expected symbologies and orientations (optional)
reader.SetExpectedSymbologies(...);
reader.SetExpectedOrientations(...);

// ...

// Read
std::vector<EMailBarcode> codes = reader.Read(srcImage);

// Retrieve the data included in found mail barcodes
for (unsigned int index= 0; index < codes.size(); index++)
```

```
{
    std::string text = codes[index].GetText();
    std::vector<EStringPair> components = codes[index]. GetComponentStrings();
}
```

6.3. EasyMatrixCode

Automatic Reading

```
//////////////////////////////////////////////////////////////
// This code snippet shows how to read a data matrix code  //
// and retrieve the decoded string.                        //
//////////////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// Matrix code reader constructor
EMatrixCodeReader reader;

// Matrix code constructor
EMatrixCode mxCode;

// String for the decoded information
std::string result;

// ...

// Read the source image
mxCode = reader.Read(srcImage);

// Retrieve the decoded string
result = mxCode.GetDecodedString();
```

Reading with Prior Learning

```
//////////////////////////////////////////////////////////////
// This code snippet shows how to learn a given data matrix //
// code type (except its flipping status), perform the      //
// reading and retrieve the decoded string.                 //
//////////////////////////////////////////////////////////////

// Images constructor
EImageBW8 model;
EImageBW8 srcImage;

// Matrix code reader constructor
EMatrixCodeReader reader;

// Matrix code constructor
EMatrixCode mxCode;

// String for the decoded information
std::string result;

// ...

// Tell the reader not to take the flipping into account when learning
reader.SetLearnMaskElement(ELearnParam_Flipping, false);

// Learn the model
reader.Learn(model);
```

```
// Read the source image
mxCode = reader.Read(srcImage);
```

```
// Retrieve the decoded string
result = mxCode.GetDecodedString();
```

Advanced Tuning of the Search Parameters

```
//////////////////////////////////////////////////////////
// This code snippet shows how to explicitly specify the data //
// matrix code logical size and family, perform the reading   //
// and retrieve the decoded string.                           //
//////////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;
```

```
// Matrix code reader constructor
EMatrixCodeReader reader;
```

```
// Matrix code constructor
EMatrixCode mxCode;
```

```
// String for the decoded information
std::string result;
```

```
// ...
```

```
// Remove the default logical sizes
reader.GetSearchParams().ClearLogicalSize();
```

```
// Add the 15x15 and 17x17 logical sizes
reader.GetSearchParams().AddLogicalSize(ELogicalSize__15x15);
reader.GetSearchParams().AddLogicalSize(ELogicalSize__17x17);
```

```
// Remove the default families
reader.GetSearchParams().ClearFamily();
```

```
// Add the ECC050 family
reader.GetSearchParams().AddFamily(EFamily_ECC050);
```

```
// Read the source image
mxCode = reader.Read(srcImage);
```

```
// Retrieve the decoded string
result = mxCode.GetDecodedString();
```

Retrieving Print Quality Grading

```
//////////////////////////////////////////////////////////
// This code snippet shows how to read a data matrix code //
// and retrieve its print quality grading.                //
//////////////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;
```

```
// Matrix code reader constructor
EMatrixCodeReader reader;
```

```
// Matrix code constructor
EMatrixCode mxCode;
```

```
// ...
```

```
// Enable grading computation
reader.SetComputeGrading(TRUE);
```

```
// Read the source image
mxCode = reader.Read(srcImage);
```

```
// Retrieve the print quality grading
int axialNonUniformityGrade= mxCode.GetAxialNonUniformityGrade();
int contrastGrade= mxCode.GetContrastGrade();
int printGrowthGrade= mxCode.GetPrintGrowthGrade();
int unusedErrorCorrectionGrade= mxCode.GetUnusedErrorCorrectionGrade();
```

6.4. EasyQRCode

Automatic Reading of a QR Code

```
///////////////////////////////////////////////////
// This code snippet shows how to read a QR code  //
// and retrieve the decoded data.                 //
///////////////////////////////////////////////////
```

```
// Image constructor
EImageBW8 srcImage;
```

```
// QR code reader constructor
EQRCodeReader reader;
```

```
// ...
```

```
// Set the source image
reader.SetSearchField(srcImage);
```

```
// Read
std::vector<EQRCode> qrCodes = reader.Read();
```

```
// Retrieve the data of the first QR code found if
// one was found and decoding went ok
if ((qrCodes.size() > 0) &&
   (qrCodes[0].GetUnusedErrorCorrection() >= 0))
{
    EQRCodeDecodedStream stream = qrCodes[0].GetDecodedStream();
}
```

Retrieving Information of a QR Code

```
///////////////////////////////////////////////////
// This code snippet shows how to read a QR code  //
// and retrieve the associated information.        //
///////////////////////////////////////////////////
```

```
// Image constructor
EImageBW8 srcImage;
```

```
// QR code reader constructor
EQRCodeReader reader;
```

```
// ...
```

```
// Set the source image
reader.SetSearchField(srcImage);
```

```
// Read
std::vector<EQRCode> qrCodes = reader.Read();
```

```
// Retrieve version, model and position information
// of the first QR code found, if one was found
if (qrCodes.size() > 0)
{
    int version = qrCodes[0].GetVersion();
```

```
    EQRCodeModel model = qrCodes[0].GetModel();
    EQRCodeGeometry geometry = qrCodes[0].GetGeometry();
}
```

Decoding the First QR Code Detected

```
/////////////////////////////////////////////////
// This code snippet shows how to decode a QR code //
// from a list of detected ones.                   //
/////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// QR code reader constructor
EQRCodeReader reader;

// ...

// Set the source image
reader.SetSearchField(srcImage);

// Detect QR Codes
std::vector<EQRCodeGeometry> qrCodeGeometries = reader.Detect();

// Decode first QR Code
EQRCode qrCode = reader.Decode(qrCodeGeometries[0]);

// Retrieve the data from the QR Code
EQRCodeDecodedStream stream = qrCode.GetDecodedStream();
```

Tuning the Search Parameters

```
/////////////////////////////////////////////////
// This code snippet shows how to read a QR code  //
// and retrieve the decoded data after setting a  //
// number of search parameters.                   //
/////////////////////////////////////////////////

// Image constructor
EImageBW8 srcImage;

// QR code reader constructor
EQRCodeReader reader;

// ...

// Set the source image
reader.SetSearchField(srcImage);

// Set the search parameters
reader.SetMaximumVersion(7);
reader.SetMinimumIsotropy(0.9f);

// Set the searched models
std::vector<EQRCodeModel> models;
models.push_back(EQRCodeModel_Model2);
reader.SetSearchedModels(models);
```

```
// Read
std::vector<EQRCode> qrCodes = reader.Read();

// Retrieve the data of the first QR code found
EQRCodeDecodedStream stream = qrCodes[0].GetDecodedStream();
```