

Unibrain

# Fire-iX SDK manual

Programmer's Manual & Reference

# Table of Contents

<b>TABLE OF CONTENTS</b> .....	<b>2</b>	<i>Resolution property</i> .....	38
<b>INTRODUCTION</b> .....	<b>4</b>	<i>IsUserDefined property</i> .....	39
<b>TECHNICAL DETAILS</b> .....	<b>5</b>	<i>IsCurrent property</i> .....	39
ARCHITECTURE.....	5	<i>Description property</i> .....	40
PERFORMANCE .....	5	<i>Identifier property</i> .....	40
INSTALLATION.....	6	<i>Fixed property</i> .....	41
DESIGN .....	6	<i>UserDefined property</i> .....	41
<b>USAGE</b> .....	<b>7</b>	<i>RawModeOverride property</i> .....	41
SCENARIO I – SIMPLE VIDEO VIEWER .....	7	<i>Width property</i> .....	42
SCENARIO II – CONTROLLING CAMERA FEATURES .....	13	<i>Height property</i> .....	42
SCENARIO III – MANIPULATING CAPTURED FRAMES.....	16	<i>Save method</i> .....	43
SCENARIO IV – WORKING WITH A ROI (A.K.A. FORMAT 7) .	18	<i>Reload method</i> .....	43
<b>REFERENCE</b> .....	<b>20</b>	<b>FIREIXFIXEDSTREAMFORMAT</b> .....	44
<b>FIREIXPREVIEWCTRL</b> .....	20	<i>ResolutionString property</i> .....	44
<i>CreateManager method</i> .....	20	<i>FrameRateString property</i> .....	44
<i>AttachCamera method</i> .....	20	<i>FrameRate property</i> .....	44
<i>LeftClick event</i> .....	21	<i>Width property</i> .....	45
<i>RightClick event</i> .....	21	<i>Height property</i> .....	45
<b>FIREIGUID</b> .....	22	<i>PacketSize property</i> .....	46
<i>Byte property</i> .....	22	<i>PacketsPerFrame property</i> .....	46
<i>ToString method</i> .....	22	<i>Description property</i> .....	47
<i>FromString method</i> .....	22	<i>IsFrameRateSupported method</i> .....	47
<b>FIREIXMANAGER</b> .....	24	<i>Save method</i> .....	48
<i>GetNumOfConnectedCameras method</i> .....	24	<i>Reload method</i> .....	48
<i>SelectCamera method</i> .....	24	<b>FIREIXUSERDEFINEDSTREAMFORMAT</b> .....	49
<i>GetCameraFromIndex method</i> .....	25	<i>Left property</i> .....	49
<i>GetCameraFromGUID method</i> .....	26	<i>Right property</i> .....	49
<i>UseDirectShow method</i> .....	27	<i>Top property</i> .....	49
<b>FIREIXREGISTER</b> .....	28	<i>Bottom property</i> .....	50
<i>Bit property</i> .....	28	<i>Width property</i> .....	50
<i>Field property</i> .....	28	<i>Height property</i> .....	51
<i>FieldLen property</i> .....	29	<i>MaxWidth property</i> .....	51
<i>SwapEndian method</i> .....	29	<i>MaxHeight property</i> .....	51
<b>FIREIXTRIGGER</b> .....	30	<i>HorizontalPositionUnit property</i> .....	52
<i>AbsControl property</i> .....	30	<i>VerticalPositionUnit property</i> .....	52
<i>Enabled property</i> .....	30	<i>WidthUnit property</i> .....	52
<i>Polarity property</i> .....	31	<i>HeightUnit property</i> .....	53
<i>Source property</i> .....	31	<i>MaxPacketSize property</i> .....	53
<i>Value property</i> .....	32	<i>PacketSizeUnit property</i> .....	53
<i>Mode property</i> .....	32	<i>PacketSize property</i> .....	54
<i>IsSupported property</i> .....	33	<i>Description property</i> .....	54
<i>HasAbsolute property</i> .....	33	<i>SetROI method</i> .....	55
<i>CanRead property</i> .....	33	<i>IsValid method</i> .....	55
<i>HasOnOff property</i> .....	34	<i>Save method</i> .....	56
<i>HasPolarity property</i> .....	34	<i>Reload method</i> .....	56
<i>CanReadRaw property</i> .....	34	<b>ENUMFIREIXSTREAMFORMATS</b> .....	57
<i>Parameter property</i> .....	35	<i>Reset method</i> .....	57
<i>IsSourceSupported method</i> .....	35	<i>Next method</i> .....	57
<i>IsModeSupported method</i> .....	35	<b>FIREIXFEATURE</b> .....	58
<i>Reload method</i> .....	36	<i>Name property</i> .....	58
<i>PullSoftwareTrigger method</i> .....	36	<i>IsSupported property</i> .....	58
<i>Save method</i> .....	36	<i>HasAbsolute property</i> .....	59
<b>FIREIXSTREAMFORMAT</b> .....	37	<i>HasOnePush property</i> .....	59
<i>PixelFormatString property</i> .....	37	<i>CanRead property</i> .....	60
<i>PixelFormat property</i> .....	37	<i>HasOnOff property</i> .....	60

<i>HasAuto property</i> .....	60	<i>FriendlyName property</i> .....	79
<i>HasManual property</i> .....	61	<i>StreamFormat property</i> .....	79
<i>MinValue property</i> .....	61	<i>Trigger property</i> .....	79
<i>MaxValue property</i> .....	61	<i>AutoExposure property</i> .....	80
<i>Absolute property</i> .....	62	<i>Shutter property</i> .....	80
<i>Enabled property</i> .....	62	<i>Gain property</i> .....	81
<i>AutoMode property</i> .....	63	<i>Iris property</i> .....	81
<i>Value property</i> .....	63	<i>ColorUB property</i> .....	81
<i>HasSoftAbsolute property</i> .....	64	<i>ColorVR property</i> .....	82
<i>SoftAbsolute property</i> .....	64	<i>Hue property</i> .....	82
<i>ValueString property</i> .....	64	<i>Saturation property</i> .....	82
<i>MinValueString property</i> .....	65	<i>Focus property</i> .....	83
<i>MaxValueString property</i> .....	65	<i>Zoom property</i> .....	83
<i>Unit property</i> .....	66	<i>Brightness property</i> .....	83
<i>AbsoluteValue property</i> .....	66	<i>Sharpness property</i> .....	84
<i>MinAbsoluteValue property</i> .....	67	<i>Gamma property</i> .....	84
<i>MaxAbsoluteValue property</i> .....	67	<i>Feature property</i> .....	84
<i>Reload method</i> .....	68	<i>RawConversion property</i> .....	85
<i>OnePush method</i> .....	68	<i>Register property</i> .....	85
ENUMFIREIXFEATURES.....	69	<i>CommandRegister property</i> .....	86
<i>Reset method</i> .....	69	<i>Icon property</i> .....	86
<i>Next method</i> .....	69	<i>NumOfMemoryPresets property</i> .....	87
FIREIXFRAME .....	70	<i>SelectStreamFormat method</i> .....	87
<i>GetPixel method</i> .....	70	<i>AttachPreviewCtrl method</i> .....	89
<i>SetPixel method</i> .....	70	<i>Run method</i> .....	90
<i>GetRGB method</i> .....	71	<i>Stop method</i> .....	90
<i>SetRGB method</i> .....	71	<i>IsRunning method</i> .....	91
<i>SaveToFile method</i> .....	72	<i>GetStreamFormatsEnumerator method</i> .....	91
<i>FlipHorizontally method</i> .....	72	<i>DisplayProperties method</i> .....	91
<i>FlipVertically method</i> .....	72	<i>IsFeatureSupported method</i> .....	92
<i>Negative method</i> .....	73	<i>GetFeaturesEnumerator method</i> .....	93
<i>ToPicture method</i> .....	73	<i>GetCurrentResolution method</i> .....	94
<i>DrawLine method</i> .....	73	<i>GetCameraPhoto method</i> .....	94
<i>DrawString method</i> .....	74	<i>SaveToMemory method</i> .....	94
<i>DrawRectangle method</i> .....	74	<i>LoadFromMemory method</i> .....	95
<i>DrawLineRGB method</i> .....	75	<i>SaveToXML method</i> .....	95
<i>DrawStringRGB method</i> .....	75	<i>LoadFromXML method</i> .....	96
<i>DrawRectangleRGB method</i> .....	76	<i>RetrieveStreamFormat method</i> .....	96
FIREIXCAMERA .....	77	<i>RetrieveStreamFormatFromIdentifier method</i> ...	97
<i>BufferMode property</i> .....	77	<i>FrameReceived event</i> .....	97
<i>GUID property</i> .....	77	<i>BufferReceived event</i> .....	97
<i>Vendor property</i> .....	78	<i>DeviceRemoved event</i> .....	98
<i>Model property</i> .....	78		
<i>Serial property</i> .....	78		

## Introduction

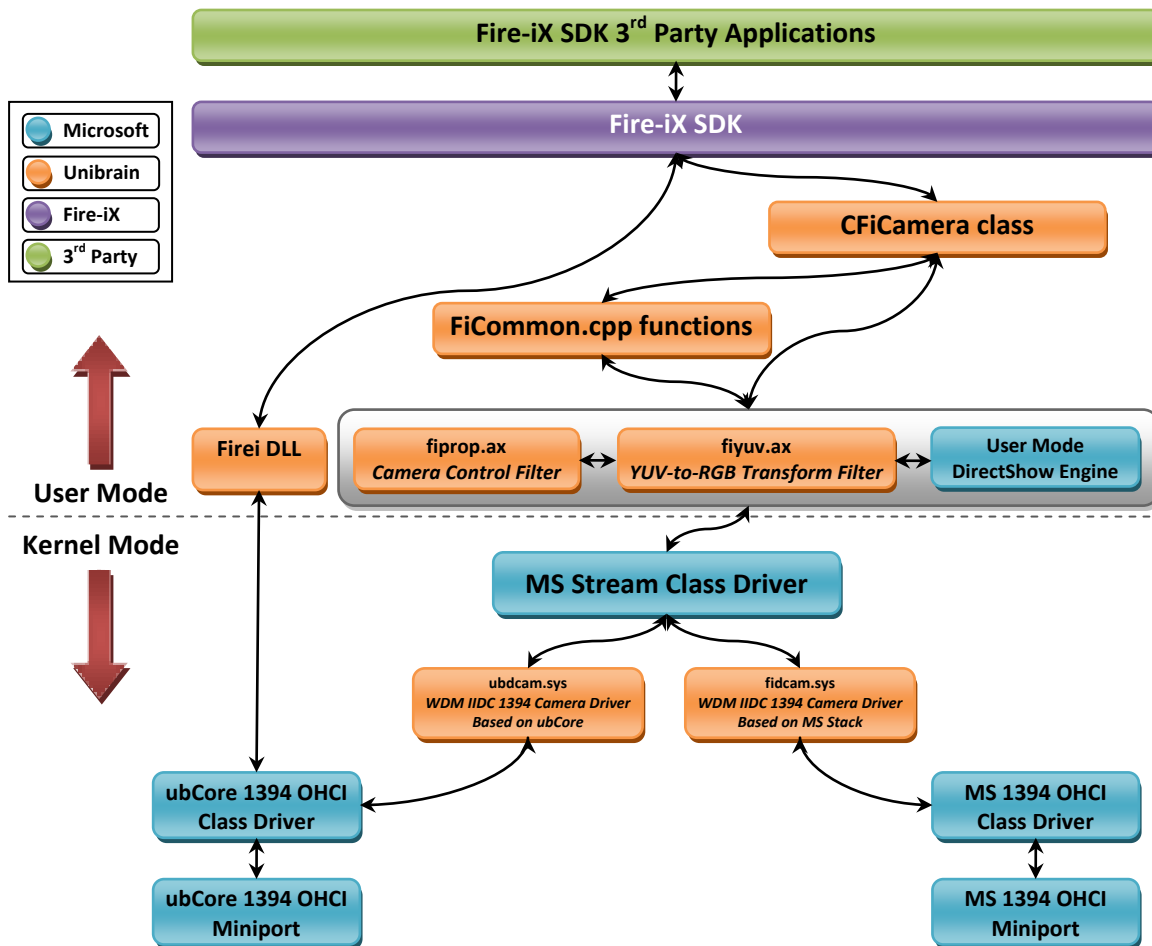
The Fire-iX SDK is the latest addition to ubCore™ and the Unibrain APIs. It provides an entirely new and comprehensive way to interact with multiple IEEE1394 IIDC compatible cameras, including video and feature manipulation, direct register access, etc. Since it is based on Microsoft™ ActiveX™ technology, it is language and development environment agnostic, integrating with all the languages with ActiveX and COM support.

The Fire-iX SDK can operate in two modes; ubCore and DirectShow. The SDK feature set is almost identical and transparent under both modes of operation. It does not require familiarity with either of the two SDKs, so it is an ideal fit in situations where the programmer does not need/want to delve into IEEE1394 or IIDC details in order to perform simple camera tasks, such as setting a few features and capturing a few frames. This is only on the surface though; the more powerful features are still there for anyone to use, they are just not required for the simpler tasks.

# Technical Details

## Architecture

The following illustration shows how the various Unibrain APIs interact with each other, and how they are abstracted from the programmer by using the Fire-iX SDK.



## Performance

The performance of the program at runtime depends on which underlying API is selected (Firei.dll, DirectShow/ubCore or DirectShow/MS Stack). There is no specific set of circumstances where selecting one API over another will produce better results. The ease of changing between all three, which requires minimal changes<sup>1</sup>, allows the programmer to test through all three different cases and compare the performance. Please keep in mind however, the performance using the MS Stack DirectShow drivers will be affected by the MS 1394.sys driver, which is known to have various issues with popular operating systems (Windows XP Service Pack 2 and Windows Vista included).

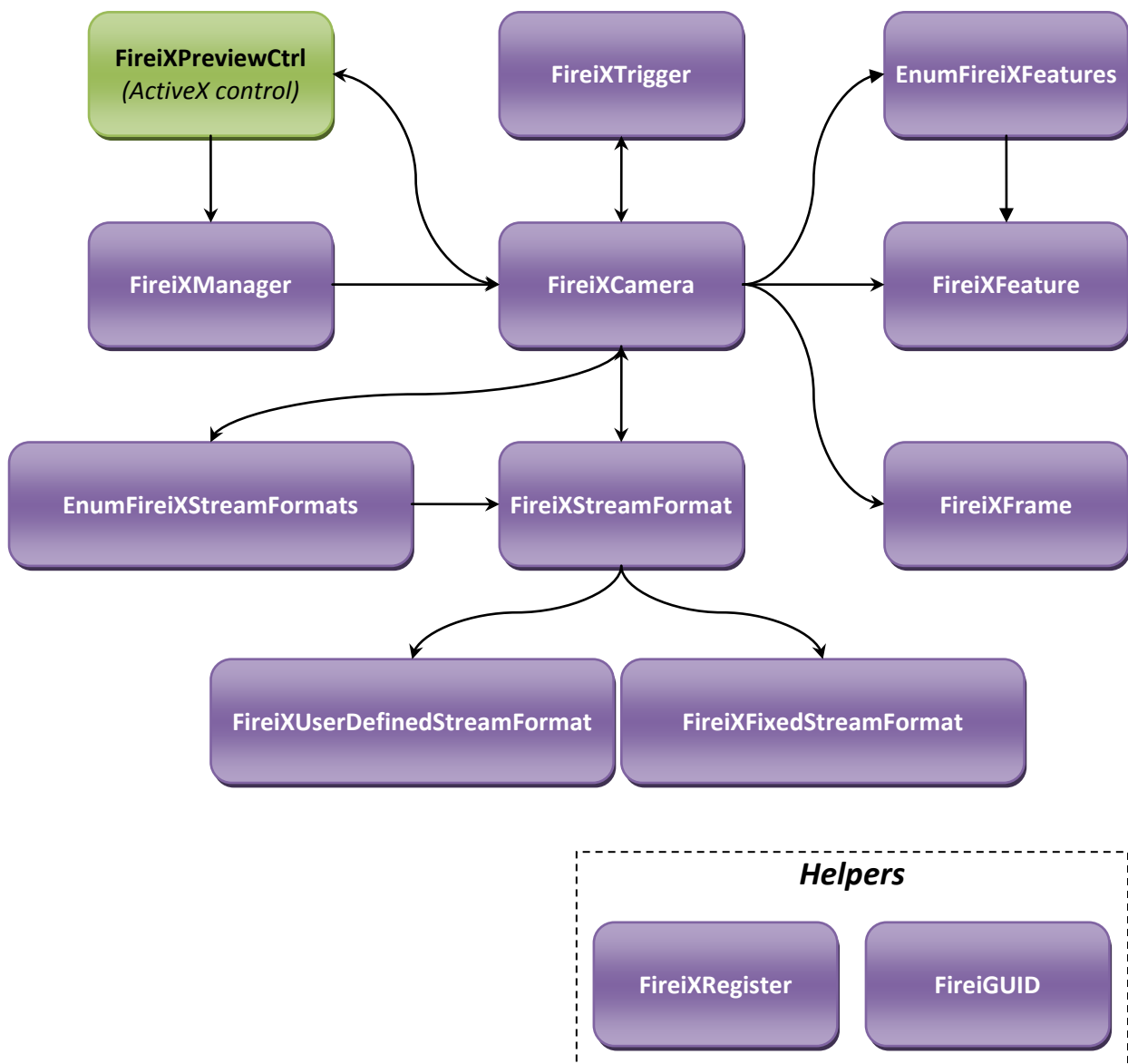
<sup>1</sup> One line of code needed to choose between Firei.dll and DirectShow, no code difference to choose between DirectShow/ubCore and MS Stack (ubSwitch is used in the latter case).

## Installation

No specific installation of Fire-iX SDK is necessary. It is included with both ubCore Setup and Firei MS Stack Setup. The FireiX.dll module (which contains the full functionality of Firei-X SDK) is copied and registered with the system automatically during these installations. It is therefore necessary to have either ubCore or Fire-i MS Stack installed and in working order for Fire-iX SDK to operate.

## Design

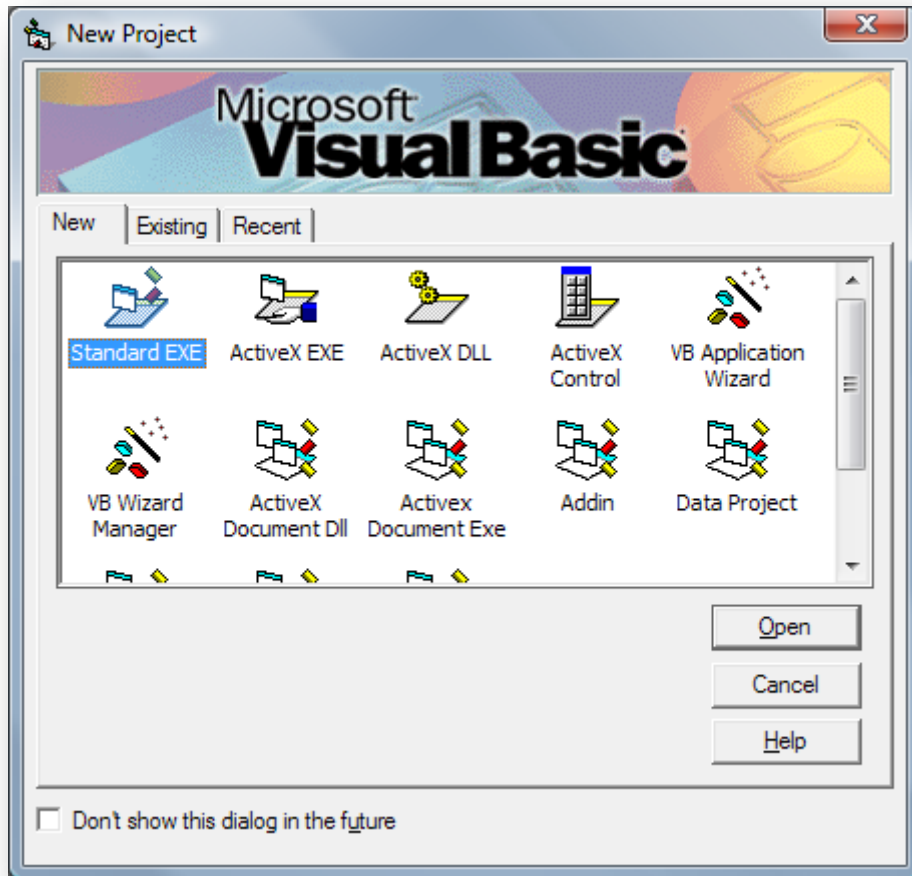
The Fire-iX SDK is designed as an ActiveX control, bundled with a few COM objects that help control the action, each with its own interface, properties and methods. The programmer either begins with the control (for use on a GUI application) and then uses it to create the other COM objects, or directly constructs the COM objects (for use on a command-line based application), omitting the ActiveX control altogether. The following diagram attempts to depict how the various COM objects interact with the ActiveX control and with each other:



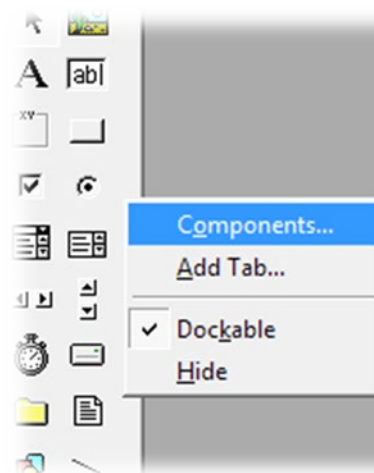
# Usage

## Scenario I – Simple video viewer

Let's start Visual Basic 6.0. Create a new empty project, of the "Standard EXE" variety:

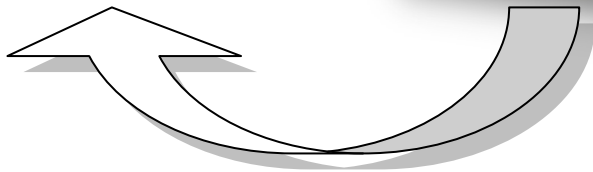
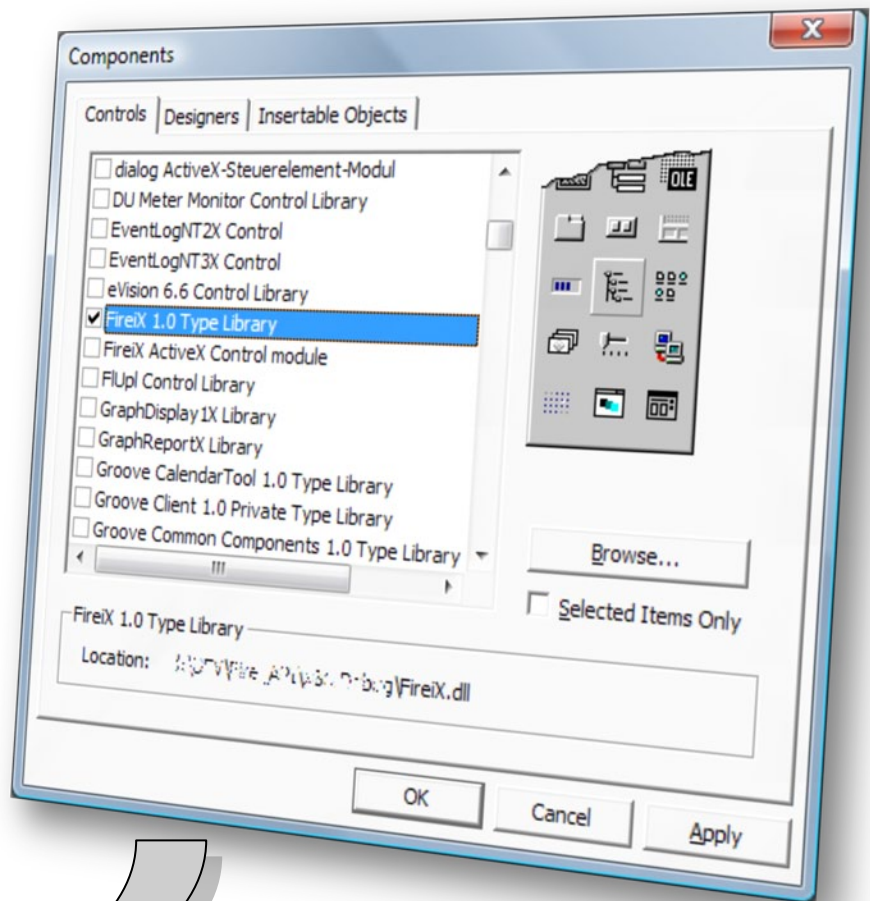


This should produce a new empty project, with a single form (empty as well). Now, right-click on the Components bar and select "Components...":

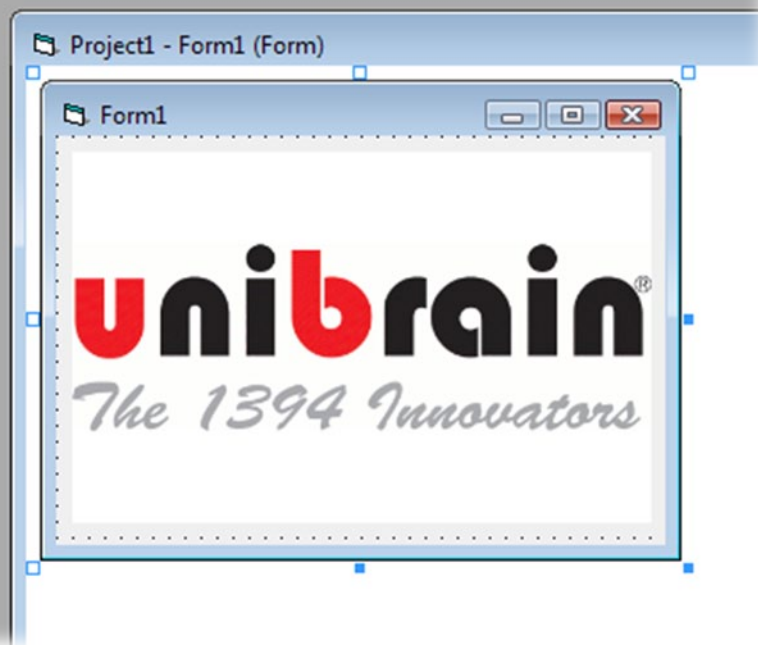


This will bring up the components selection dialog of VB. In the "Controls" tab, select the "FireiX 1.0 Type Library" and click on OK:

This will add the Fire-iX FireiXPreviewCtrl object selector, which can be used for video display:



Click on the "ub" tool, and create a window for video display on the empty form:



VB automatically sets the name of the object to be "FireiXPreviewCtrl1", which for our purpose will do nicely.

Now it's time to edit the form code. First, we need two global variables, so we add them at the very top:

```
Dim Manager As FireiXManager
Dim Camera As FireiXCamera
```

The FireiXManager object will be used to construct our camera object. The FireiXCamera object exposes all the features that our camera supports. We need to initialize our Manager object before using it. This can be done either by calling:

```
Set Manager = New FireiXManager
```

or by using the factory contained with the ActiveX control, FireiXPreviewCtrl, like:

```
Set Manager = FireiXPreviewCtrl1.CreateManager
```

The two methods of constructing a FireiXManager object will produce the exact same result, so choosing between the two is a matter of personal preference.

We override the Form\_Load event to place our initialization code:

```
Private Sub Form_Load()
    Set Manager = FireiXPreviewCtrl1.CreateManager
End Sub
```

Now we need to construct our camera object. FireiXManager provides two methods for doing just that; GetCameraFromIndex and GetCameraFromGUID. The former is useful when a) we have only one camera connected to our system or b) we want to programmatically iterate between all the connected cameras, and select the one we want<sup>2</sup>. To simply construct the first (or only) camera connected to our system, a call

```
Set Camera = Manager.GetCameraFromIndex(0)
```

will suffice. To iterate between all cameras, we can use GetNumOfConnectedCameras and proceed with a regular for-loop:

```
Dim NumOfConnectedCameras As Integer
NumOfConnectedCameras = Manager.GetNumOfConnectedCameras
For i = 0 To NumOfConnectedCameras - 1
    Set Camera = Manager.GetCameraFromIndex(i)
    If Camera.Vendor = "Unibrain" Then Exit For
Next
If Camera.Vendor <> "Unibrain" Then
    Unload Me
Exit Sub
End If
```

---

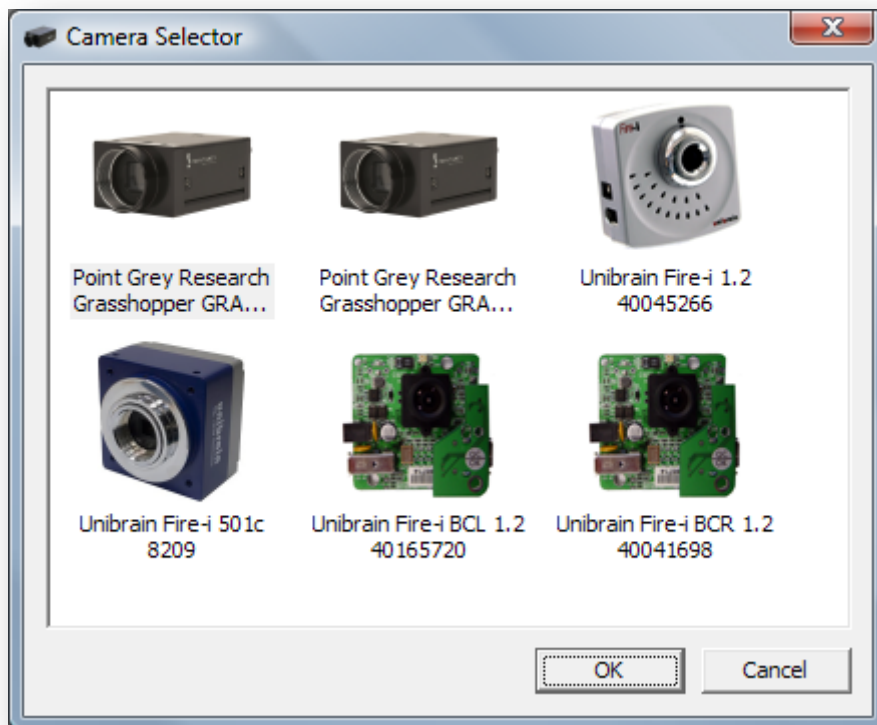
<sup>2</sup> Please keep in mind that the order in which the cameras will be indexed is non-deterministic. It can be considered to be random even on the same system with the exact same configuration between reboots.

The above code fragment will iterate through the connected cameras, and stop at the first camera that is made by Unibrain. With the additional check at the end, it will effectively select the first available Unibrain camera.

If the camera GUID (unique identifier) is known beforehand, we can use the `GetCameraFromGUID` method of `FireiXManager`. We construct a `FireiGUID` object, initialize it with our known GUID, and then construct the camera with `GetCameraFromGUID`:

```
Dim GUID As FireiGUID
Set GUID = New FireiGUID
GUID.FromString "08:14:43:61:02:63:0A:D2"
Set Camera = Manager.GetCameraFromGUID(GUID)
Set GUID = Nothing
```

Additionally, `FireiXManager` allows choosing a camera using a GUI through the `SelectCamera` method. This method will present a camera selection dialog, and if the user selects one, it will return the camera's GUID:



For simplicity, we will follow this path in this example. So, after our `FireiXManager` construction, we add:

```
Dim GUID As FireiGUID
If Manager.SelectCamera(GUID) = False Then
Unload Me
Exit Sub
End If
```

If the user didn't select a camera, the `SelectCamera` call will return `False`, in which case we exit the application. The `FireiGUID` object is constructed inside the `SelectCamera` method, and it is then used to create the camera object:

```
Set Camera = Manager.GetCameraFromGUID(GUID)
Set GUID = Nothing
```

Next, we need to connect the `Camera` object to our `FireiXPreviewCtrl` object. This is done with a single call, either:

```
FireiXPreviewCtrl1.AttachCamera Camera
```

Or

```
Camera.AttachPreviewCtrl FireiXPreviewCtrl1
```

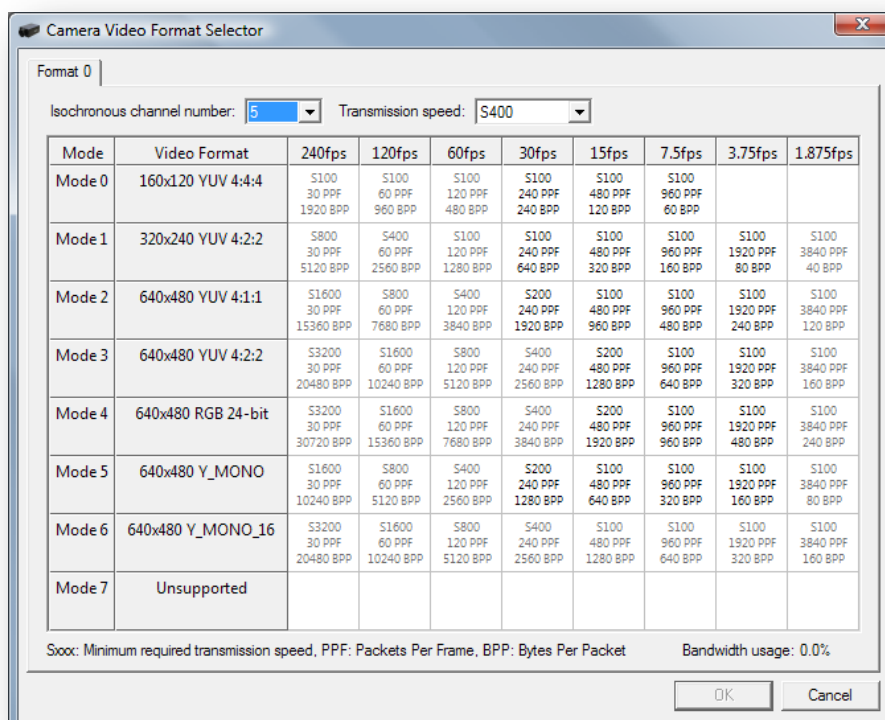
Again, this will produce the exact same result, it is a matter of programming preference. Effectively it means that any video this camera will produce will end up being shown in this preview control.

Now we need to select which streaming format our camera will run at. This can be done programmatically by either accessing the currently selected streaming mode (through the `StreamFormat` property of `FireiXCamera`), or by accessing the supported streaming formats enumerator, through the `GetStreamFormatsEnumerator` method. For the purposes of our simple example though, we will go down a different path, the GUI-driven one.

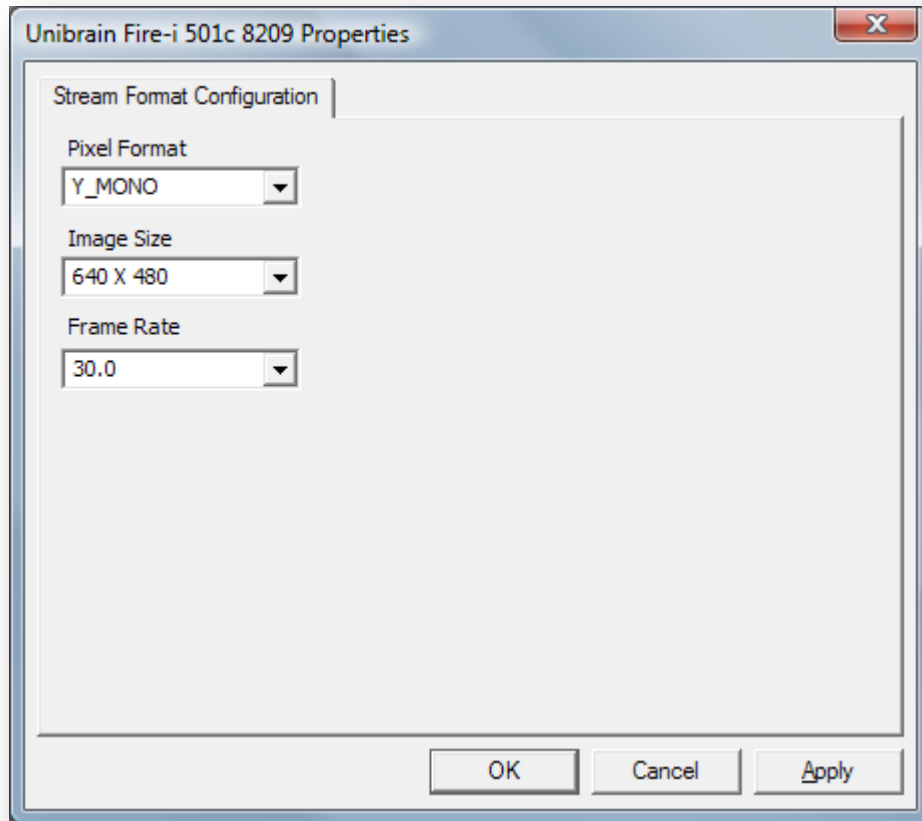
We add a call to `SelectStreamFormat`:

```
Camera.SelectStreamFormat
```

This will allow selecting and setting a streaming format through the API provided GUI:



The available choice will of course vary depending on which camera this is called upon; only supported formats will be presented. Also, the above selection dialog shown is the one provided by the Firei.dll API. Using the DirectShow API will show a format selection dialog similar to:



Now that the streaming format is selected, the only part remaining is actually starting the camera. This requires a simple call:

```
Camera.Run
```

Some cleanup is also necessary though, according to best programming practices. Before exiting, we'll stop the camera and destroy our two constructed objects. This can be done at the Form\_Unload event:

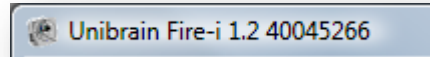
```
Private Sub Form_Unload(Cancel As Integer)
    If Not Camera Is Nothing Then Camera.Stop
    Set Camera = Nothing
    Set Manager = Nothing
End Sub
```

Running the program will show live video on our control on the form, after selecting the desired camera and streaming format. If only one camera is connected on the system, it is selected without presenting a selection dialog at all (and the SelectCamera method returns True immediately).

A nice touch to our program: Set the caption and the form icon to match our selected camera:

```
Me.Caption = Camera.FriendlyName
Me.Icon = Camera.Icon
```

This will change the form caption to look like:



By default, the Fire-iX SDK will use the Firei.dll API internally. If for some reason DirectShow is preferred, it can be arranged with a single call, placed immediately after the FireiXManager object construction:

```
Manager.UseDirectShow
```

This call can be used only once in our program. Switching between underlying APIs on-the-fly at runtime is not supported. No other change in our code is necessary, so simply commenting in and out this line will switch between the two underlying APIs for testing purposes. With DirectShow enabled and ubCore installed, by using ubSwitch (the ubCore tool) we can further toggle between ubCore DirectShow support and MS-Stack DirectShow support.

## ***Scenario II – Controlling camera features***

Now suppose we need to change some camera features, like brightness or shutter speed. Again, there are more than one ways of achieving that. Each feature for example can be accessed individually through its FireiXFeature derived object; we can access that object by a simple direct call to the corresponding camera property, like:

```
Dim Feature As FireiXFeature
Set Feature = Camera.Shutter
```

Another way to access a feature would be by name, using the Feature property of FireiXCamera:

```
Dim Feature As FireiXFeature
Set Feature = Camera.Feature("Shutter")
```

Suppose we want to change the shutter value to its maximum supported. Now that we have the Shutter feature object, this can be done by:

```
If Feature.IsSupported Then
    Feature.AutoMode = False
    Feature.Value = Feature.MaxValue
End If
Set Feature = Nothing
```

Note that the above code fragment checks whether the camera supports the feature before accessing any of its properties. Failure to do so would produce an error if the feature is not supported. Also, if the Shutter feature is set to Auto, we'll need to turn that off before setting the value, or an error will occur as well.

Yet another way to access the camera features is through the EnumFireiXFeatures enumerator provided by FireiXCamera. This allows iterating through all the supported camera features, like:

```
Dim Feature As FireiXFeature
```

```

Dim FeaturesEnum As EnumFireiXFeatures
Set FeaturesEnum = Camera.GetFeaturesEnumerator(True, fgAll)
Do While FeaturesEnum.Next(Feature)
If Feature.HasAuto Then Feature.AutoMode = True
Loop
Set FeaturesEnum = Nothing
Set Feature = Nothing

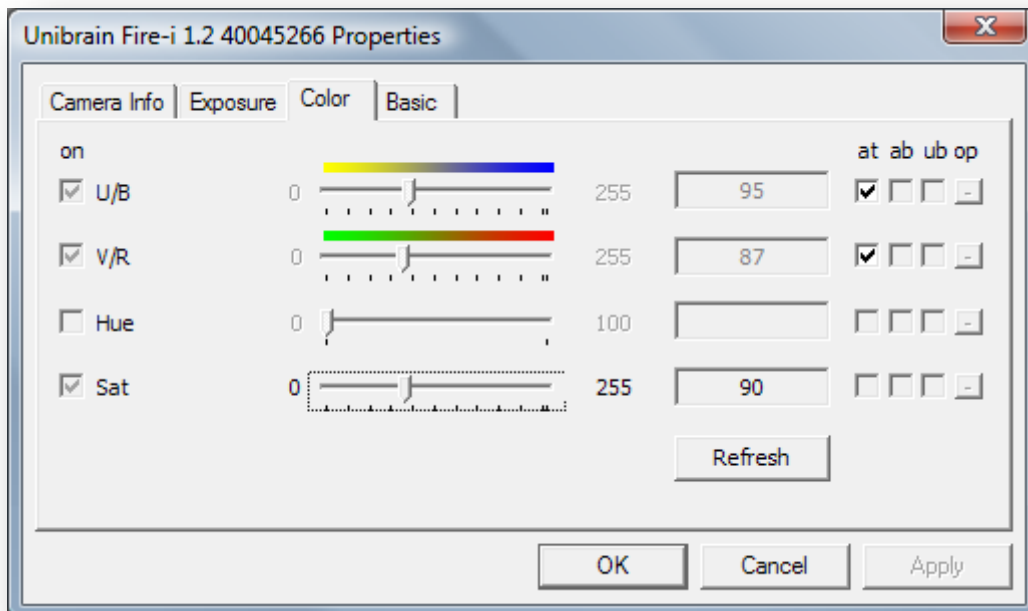
```

The above example iterates through all the supported features (for additional information on `GetFeaturesEnumerator`, see the reference section of this manual) of the camera, checks whether they support Auto setting and if they do, sets it.

A GUI-driven way to manipulate the camera features is also provided. A call to the `DisplayProperties` method of `FireiXCamera` will suffice:

```
Camera.DisplayProperties
```

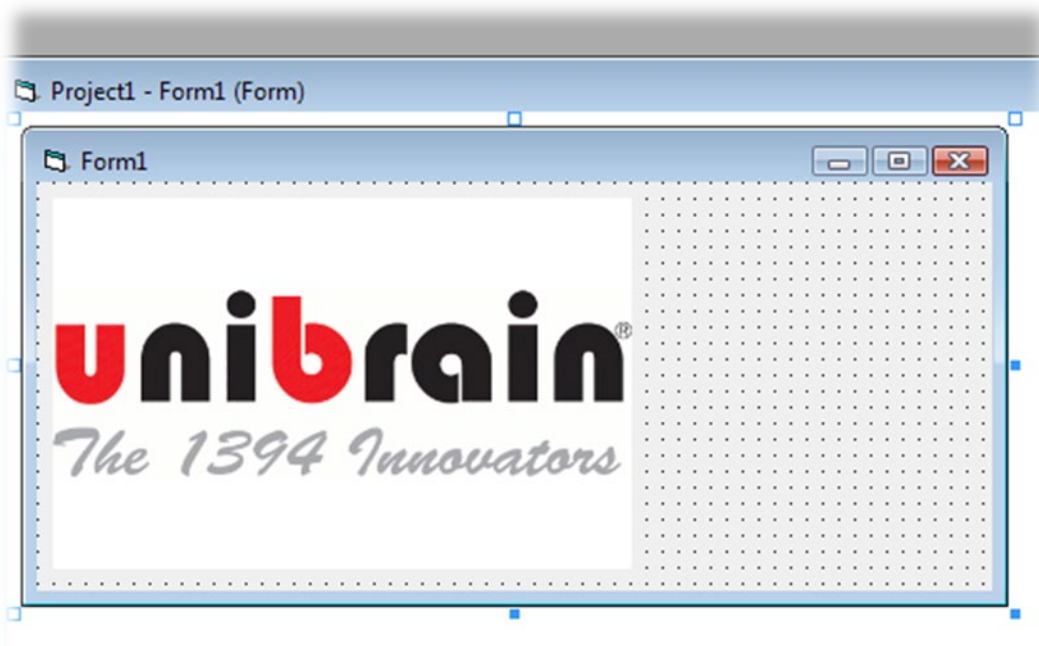
This brings up the camera properties sheet:



**NOTE:** The call to `DisplayProperties` returns immediately after opening the dialog. Therefore it is possible to manipulate the camera features while observing the results in the preview window. There is no additional functionality that can be accessed through the GUI feature manipulator; everything that can be done through the GUI, can also be done programmatically. For example, the Shutter feature discussed above resides in the “Exposure” tab. The “AutoMode” setting is the one marked “at” with a checkbox. The slider control corresponds to the value setting.

For the purposes of our example, we’ll add a listbox to our Scenario I sample application, containing the names of all the features that are supported by the camera, and a button that brings up the camera properties.

Firstly, we enlarge our form, to make way for the new controls:



Then, we add the two controls. It is also a good idea to change their corresponding variable names to FeatureList and Properties respectively:



The best time to populate the listbox would be immediately after we constructed our FireiXCamera object. As discussed above, iterating through the supported features is done with the EnumFireiXFeatures enumerator:

```
Dim Feature As FireiXFeature
Dim FeaturesEnum As EnumFireiXFeatures
Set FeaturesEnum = Camera.GetFeaturesEnumerator(True, fgAll)
```

```

Do While FeaturesEnum.Next(Feature) = True
FeaturesList.AddItem Feature.Name
Loop
Set FeaturesEnum = Nothing
Set Feature = Nothing

```

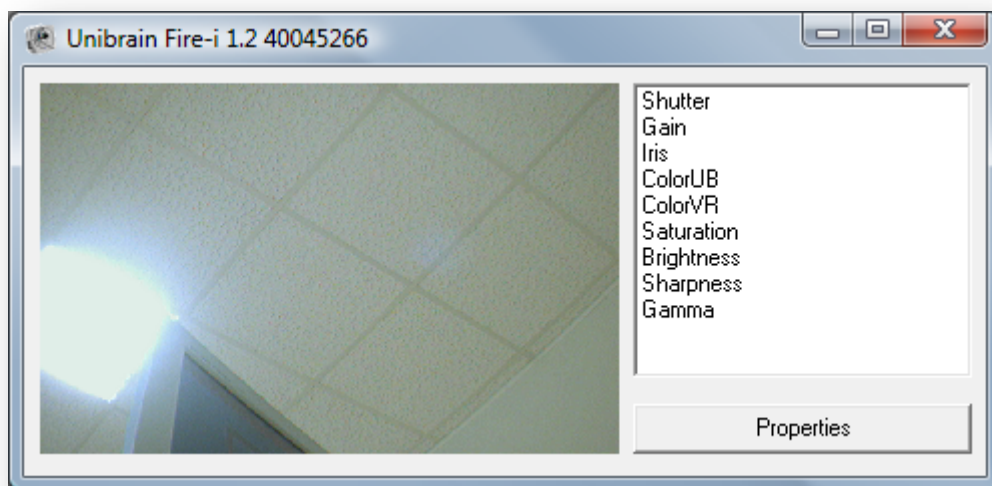
This takes care of the listbox population. Now for the properties button, we override the Properties\_Click event:

```

Private Sub Properties_Click()
If Not Camera Is Nothing Then Camera.DisplayProperties
End Sub

```

Running the program produces the desired result:



Most cameras will reset to their default feature values if powered off. The SDK provides two ways to save and restore them; to/from an XML file through the LoadFromXML and SaveToXML methods of FireiXCamera and, if the camera supports memory presets, through the SaveToMemory and LoadFromMemory methods. Note that the XML formatting of the file is compatible with the other tools of ubCore. So for instance, the programmer can use Fire-i Application or FireIIDC to select and then save a feature set, and then load it programmatically in a Fire-iX program.

### ***Scenario III – Manipulating captured frames***

Now that we have our simple viewer program up and running, watching the live video preview, let's take it a step further: we can manipulate the camera frames as they arrive.

In order to do that, we need to override the FrameReceived event of FireiXCamera. It is important to declare our global camera variable with the " WithEvents " designation if event handling is required, so we change the Camera declaration to:

```
Dim WithEvents Camera As FireiXCamera
```

Now we can safely override the FrameReceived event:

```
Private Sub Camera_FrameReceived(ByVal Frame As FireiXLibCtl.IFireiXFrame)
End Sub
```

As shown above, the event provides a FireiXFrame object for our perusal. Through this object we have absolute access to the frame buffer, pixel by pixel, with the GetPixel/SetPixel and GetRGB/SetRGB methods. The only difference between the two sets is the way the color value of the pixel is passed; in the former it is passed as a Long value representing the color, in the latter it is passed with R, G and B values separately. The same designation holds true for the other methods of FireiXFrame; whenever "RGB" is in their name, it means separate R, G and B values.

Besides direct per-pixel manipulation, a few additional methods are implemented in FireiXFrame. Suppose we'd like to draw a blue rectangular box, along the edges of the frame, and a big red X inside it. We can use the DrawRectangleRGB and DrawLineRGB methods to achieve that:

```
Dim x1, y1, x2, y2 As Integer
x1 = 10
y1 = 10
x2 = Camera.StreamFormat.Width - x1
y2 = Camera.StreamFormat.Height - y1
Frame.DrawLineRGB x1, y1, x2, y2, 255, 0, 0
Frame.DrawLineRGB x2, y1, x1, y2, 255, 0, 0
Frame.DrawRectangleRGB x1, y1, x2 - x1, y2 - y1, 0, 0, 255, False
```

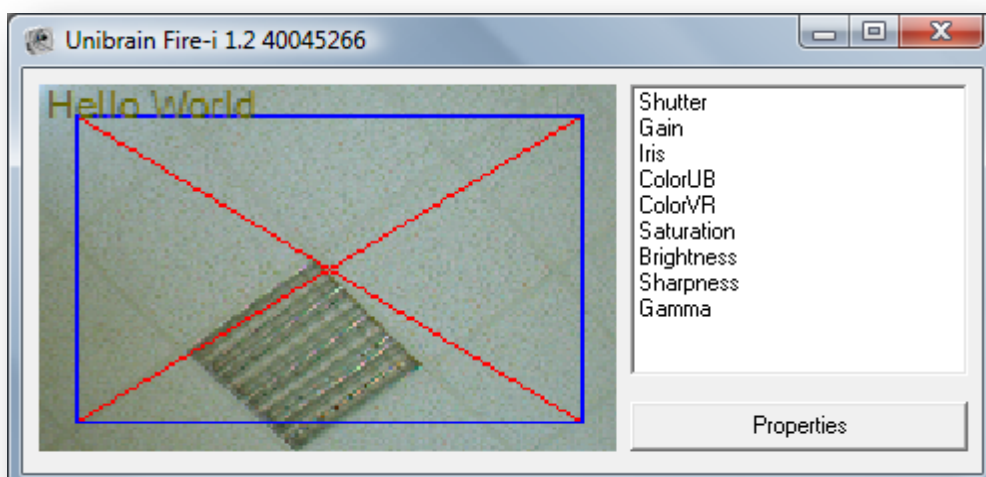
Note the way x2 and y2 are calculated: we use the Width and Height properties of the selected streaming format to ascertain the edges of our frame. The "False" in the last parameter of DrawRectangleRGB, means "empty" as opposed to "filled" box.

Now suppose we'd like to write some text on the frame as well. We can use the DrawStringRGB method for this:

```
Frame.DrawStringRGB "Hello World", 0, 0, Me.Font, 100, 100, 0
```

A yellowish color is chosen this time, using the default font of our form (Me.Font), placed at the top left corner of the frame.

This is what the outcome looks like:



**NOTE:** The aliasing of the drawn lines and text is due to the enlargement of the picture in order for it to fit in our selected preview control. The actual streaming format that was selected for the above screenshot was 160×120, which is less than the size of the window. Similarly, if the format was larger than the screen, the image would have been shrunk to fit accordingly. Also, to nitpick, there is no test made whether the image is actually larger than 20×20, as would be required for the box and lines to fit. In case the image was smaller than 20×20, an error would occur trying to call DrawLineRGB with positions out-of-bounds.

Additionally, we can save the frame at any time to a Windows BMP compatible file on disk. To do that, simply call the SaveToFile method of FireiXFrame:

```
Frame.SaveToFile "c:\1.bmp"
```

The placement of the SaveToFile call is important. If it is placed before any changes on the frame, the file will contain the frame as it is, whereas if it is placed after any changes, it will contain them too. The resulting file will have the Width × Height size as sent by the camera, not as shown on the screen. So for instance, in this case it would be 160×120, regardless of the size of the window showing the video.

**NOTE:** Please exercise caution when using the SaveToFile method. If a high frame rate streaming format is selected, especially with a high resolution, the overhead on the system saving individual uncompressed BMP files in quick succession can be extremely taxing. For example, a 1280×960×24bit image has a size on the disk of roughly 3.52MB. At 30 frames per second, the throughput required would be above 100MB per second, which is prohibitive in most real-world cases. In such instances it is preferable not to save every frame captured but every *n*-th frame to reduce overhead – or simply use a lower frame rate.

### ***Scenario IV – Working with a ROI (a.k.a. Format 7)***

Some cameras support capturing a specific ROI (Region Of Interest), through what is defined in the IIDC specification as “Format 7” or “Partial Image Format”. The Fire-iX SDK has full support for those modes of operation, by using a derivative of the FireiXStreamFormat object, called FireiXUserDefinedStreamFormat (as opposed to the FireiXFixedStreamFormat).

The SelectFormat method discussed in Scenario I allows such selection, but this time we’ll go a different way, and manipulate the streaming format programmatically.

First, we need to iterate through all available streaming formats of the camera. We do that through the EnumFireiXStreamFormats enumerator, in a similar way to iterating through the camera features:

```
Dim StreamFormat As FireiXStreamFormat
Dim StreamFormats As EnumFireiXStreamFormats
Set StreamFormats = Camera.GetStreamFormatsEnumerator
Do While StreamFormats.Next(StreamFormat) = True
If StreamFormat.IsUserDefined = True Then Exit Do
Loop
If StreamFormat Is Nothing Then
Unload Me
Exit Sub
End If
Dim w, h As Integer
w = StreamFormat.UserDefined.MaxWidth
```

```
h = StreamFormat.UserDefined.MaxHeight
StreamFormat.UserDefined.SetROI 0, 0, w, h
Camera.StreamFormat = StreamFormat
Set StreamFormat = Nothing
```

In the above code fragment, an `EnumFireiXStreamFormats` enumerator is requested from the camera, and the first User Defined stream format is selected through iteration. If the camera doesn't support any User Defined formats, at the end of the iteration the `StreamFormat` variable will be `Nothing` – in which case we exit the program. Since we ascertained that `StreamFormat` in fact is valid, and is of User Defined type, we can use the `UserDefined` property of `FireiXStreamFormat` to access the User Defined properties of the format. In this case we assign in the `w` and `h` integer variables the maximum allowed width and height of the format respectively and then set it as our selection, using the `SetROI` method. Since no complex values are set we don't check the return state of `SetROI` – if an invalid rectangle was specified as the User Defined size the `SetROI` method would return `False`. Finally, we set the now updated `StreamFormat` to the Camera, via its `StreamFormat` property – we could also have done this through the `Save` method of the `FireiXStreamFormat` itself.

In fact, the `UserDefined` property of `FireiXStreamFormat` returns a variable of type `FireiXUserDefinedStreamFormat`. In case the `FireiXStreamFormat` is `Fixed` instead of User Defined, the corresponding property is called `Fixed` accordingly and returns a variable of type `FireiXFixedStreamFormat`. Even though a `Fixed` format carries that name, it contains a single variable property – the frame rate (it can be set using its `FrameRate` property).

All formats, whether `Fixed` or User Defined, have two attributes in common: their `FireiXResolution` and their `FireiXPixelFormat`. Both of these attributes are enumerated values – in the case of `FireiXResolution`, the name is sort of a misnomer; if the stream format is User Defined it is actually valued from `resVariable` to `resVariable_7` which are roughly equivalent to the `Format_7` Modes of the camera (unlike for `Fixed` formats, which are valued from `res160x120` to `res1600x1200`). This pair of values is unique to a stream format, so it is sufficient to define the format.

Since the above situation is not always easy to work with, another way to identify a streaming format exists: a unique identifier, which can be stored in a `Long` value. This is accessed through the `Identifier` property of `FireiXStreamFormat`. This identifier can be used to retrieve this specific stream format from the camera, using its `RetrieveStreamFormatFromIdentifier` method. Keep in mind though that any user selected settings are not stored – it is the stream format that is retrieved, not its contents.

## Reference

In the following reference of all interfaces implemented, “Long” is defined as a 32-bit signed integer value, and “Integer” is defined as a 16-bit signed integer value. “Byte” on the other hand is an 8-bit unsigned integer value. “Single” is a single-precision floating-point number.

### *FireiXPreviewCtrl*

The FireiXPreviewCtrl is an ActiveX control, which purpose is to show the video output of the underlying API at work. It can be resized at will while running, and can be attached on any form type, on any language or programming environment that supports ActiveX.

Additionally, it has the following methods and events implemented in its interface:

### **CreateManager method**

#### *Prototype*

```
FireiXManager CreateManager()
```

#### *Comments*

This method is a simple factory for a FireiXManager object instance. The created object is not dependent on the lifecycle of the FireiXPreviewCtrl, it can be considered valid even when the control has been destroyed.

Note that the FireiXManager object can be directly created in any language that supports COM object creation. The CreateManager method is useful for languages that do not, or make it very difficult to do so.

#### *Visual Basic 6.0 syntax*

```
Dim Manager As FireiXManager  
Set Manager = FireiXPreviewCtrl1.CreateManager
```

#### *C++ syntax*

```
IFireiXManager* pIManager;  
HRESULT hr = pIPreviewCtrl1->CreateManager(&pIManager);
```

### **AttachCamera method**

#### *Prototype*

```
AttachCamera(IN FireiXCamera Camera)
```

#### *Comments*

With this method, an already created FireiXCamera object is attached to this FireiXPreviewCtrl. This “attachment” means in plain words, that whenever from now on the camera sends video, it will be displayed on this control.

AttachCamera can be called many times during the lifecycle of the FireiXPreviewCtrl, each time the existing attached camera (if any) is replaced with the new one. It is very conceivable that many cameras are in succession attached to a single control, if desired by the programmer.

### *Visual Basic 6.0 syntax*

```
FireiXPreviewCtrl1.AttachCamera Camera
```

### *C++ syntax*

```
HRESULT hr = pIPreviewCtrl1->AttachCamera(pICamera);
```

## **LeftClick event**

### *Prototype*

```
LeftClick(IN Long X, IN Long Y)
```

### *Comments*

This event is fired whenever a user clicks the left mouse button on the `FireiXPreviewCtrl`.

The X and Y parameters will contain the coordinates the click happened on. These coordinates are expressed relative to the `FireiXPreviewCtrl`.

### *Syntax*

Since event handlers are usually created automatically by the underlying programming environment, it would not be useful to provide sample code here.

## **RightClick event**

### *Prototype*

```
RightClick(IN Long X, IN Long Y)
```

### *Comments*

This event is fired whenever a user clicks the right mouse button on the `FireiXPreviewCtrl`.

The X and Y parameters will contain the coordinates the click happened on. These coordinates are expressed relative to the `FireiXPreviewCtrl`.

### *Syntax*

Since event handlers are usually created automatically by the underlying programming environment, it would not be useful to provide sample code here.

## **FireiGUID**

The FireiGUID object is a helper object, making the camera GUID easier to manipulate, store and retrieve.

It can be constructed directly, or through the `SelectCamera` method of `FireiXManager`. It supports a number of properties and methods through its interface:

### **Byte property**

#### *Prototype*

```
Byte Byte(IN Byte Index)
```

#### *Comments*

The `Byte` property is a read/write property, enabling the direct manipulation of any of the 8 bytes in a `FireiGUID` object.

#### *Visual Basic 6.0 syntax*

```
Dim Byte0 As Byte
Byte0 = GUID.Byte(0)           'get
GUID.Byte(0) = Byte0         'set
```

#### *C++ syntax*

```
BYTE Value;
HRESULT hr = pIGUID->get_Byte(0, &Value); //get
Hr = pIGUID->put_Byte(0, Value);         //set
```

### **ToString method**

#### *Prototype*

```
String ToString()
```

#### *Comments*

The `ToString` method converts and returns the bytes contained in the `FireiGUID` object to a regular string, with each of the 8 bytes separated with the ':' character for better clarity. Each byte is represented in 2-character hexadecimal form.

#### *Visual Basic 6.0 syntax*

```
Dim StringGUID As String
StringGUID = GUID.ToString
```

#### *C++ syntax*

```
BSTR StringGUID;
HRESULT hr = pIGUID->ToString(&StringGUID);
```

### **FromString method**

#### *Prototype*

```
FromString(IN String StringGUID)
```

### *Comments*

The `FromString` method can be used to initialize a `FireiGUID` object with an 8-byte GUID represented as a string, with each byte being represented as a 2-character hexadecimal value. The bytes can be separated by any single character (must be the all the way) or not separated at all.

The output of the `ToString` method can be fed to the `FromString` method directly if desired.

### *Visual Basic 6.0 syntax*

```
Dim StringGUID As String
StringGUID = "XX:XX:XX:XX:XX:XX:XX:XX"
GUID.FromString StringGUID
```

### *C++ syntax*

```
CString StringGUID(_T("XX:XX:XX:XX:XX:XX:XX:XX"));
HRESULT hr = pIGUID->FromString(StringGUID.AllocSysString());
```

## ***FireiXManager***

The FireiXManager object is the starting point for the selection and construction of cameras.

It itself can be constructed either directly, or through the CreateManager method of FireiXPreviewCtrl. By constructing it directly, a Fire-iX program can be completely detached from any UI, existing entirely in command line (provided no video preview is necessary, just image processing and/or manipulation).

With the use of UseDirectShow the programmer can opt to use the DirectShow API instead of the Firei.dll API. This call is entirely sufficient, no other code changes are necessary.

It has a number of methods implemented, all (except UseDirectShow) having to do with the selection and construction of FireiXCamera objects.

### **GetNumOfConnectedCameras method**

#### ***Prototype***

```
Byte GetNumOfConnectedCameras()
```

#### ***Comments***

This method will return the number of connected cameras on the bus.

Please keep in mind that since cameras can be freely plugged in and unplugged, this call can return different results at different times, even during runtime. Also, the number of cameras connected to the bus is not the same as the number of cameras directly connected to the PC running the Fire-iX program. The 1394 bus networks any connected cameras and PCs together, and the call to GetNumOfConnectedCameras will return *all* the accessible cameras from this PC.

This call is not light on system resources (it produces a lot of traffic on the 1394 bus searching for cameras) and care should be exercise when using it.

#### ***Visual Basic 6.0 syntax***

```
Dim NumOfCameras As Byte  
NumOfCameras = Manager.GetNumOfConnectedCameras
```

#### ***C++ syntax***

```
BYTE NumOfCameras;  
HRESULT hr = pIManager->GetNumOfConnectedCameras(&NumOfCameras);
```

### **SelectCamera method**

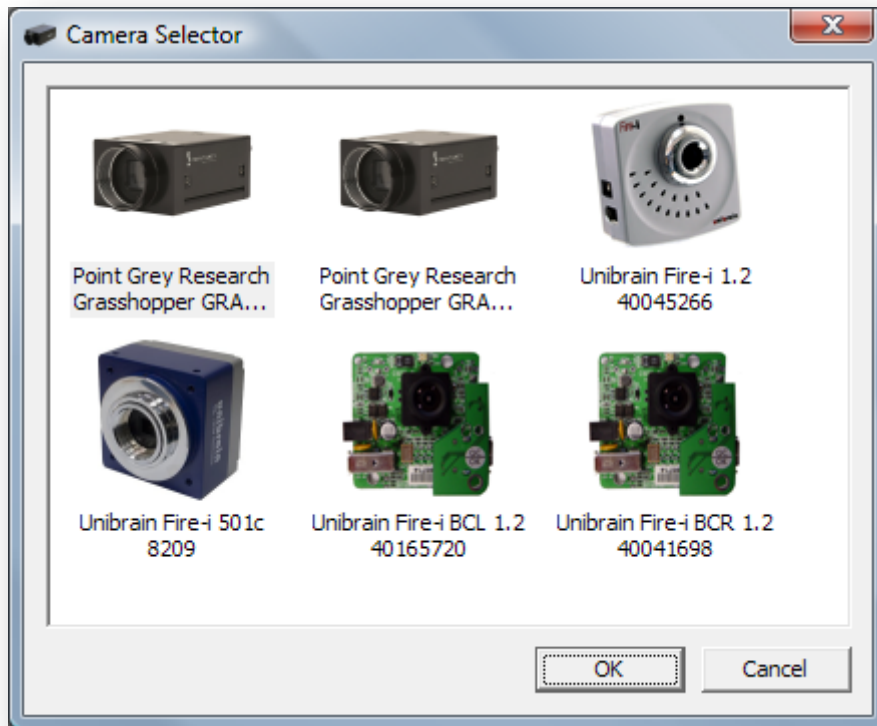
#### ***Prototype***

```
Boolean SelectCamera(OUT FireiGUID GUID)
```

#### ***Comments***

This method when called will bring up a "Camera Selector" dialog, as constructed and maintained internally by the APIs. Through this dialog, user selection of a camera is possible. The user can double-click on a camera to select it, or click on one and then "OK".

The dialog presented looks similar to:



The method returns True if a selection was made by the user and False if “Cancel” was pressed (no selection made). If the result is True, then a FireiGUID object is constructed and passed through the GUID parameter. This FireiGUID can then be used by other methods, such as the GetCameraFromGUID method.

If only one camera is connected to the system, the method returns True immediately, without presenting the selector dialog. The FireiGUID then represents the GUID of the single camera.

#### *Visual Basic 6.0 syntax*

```
Dim GUID As FireiGUID
Dim Selected As Boolean
Selected = Manager.SelectCamera(GUID)
```

#### *C++ syntax*

```
IFireiGUID* pIGUID;
VARIANT_BOOL bSelected;
HRESULT hr = pIManager->SelectCamera(&pIGUID, &bSelected);
```

### **GetCameraFromIndex method**

#### *Prototype*

```
FireiXCamera GetCameraFromIndex(IN Byte Index)
```

### Comments

This method constructs a `FireiXCamera` object and returns it, given an Index number. This number must be less than `GetNumOfConnectedCameras`, starting from 0.

It is most useful when:

- A single camera is connected (a call with 0 index is sufficient).
- The order in which the cameras are created is not important, and all camera objects must be created (a for-loop from 0 to `GetNumOfConnectedCameras - 1` is sufficient).
- A search for a specific camera, following specific criteria is required (the same for-loop, this time looking at the properties of each camera and stopping at the first that meets the criteria).

### Visual Basic 6.0 syntax

```
Dim Camera As FireiXCamera
Dim i, NumOfCameras As Byte
NumOfCameras = Manager.GetNumOfConnectedCameras
For i = 0 To NumOfCameras - 1
    Set Camera = Manager.GetCameraFromIndex(i)
    If Camera.Vendor = "Unibrain" Then Exit For
Next
```

### C++ syntax

```
IFireiXCamera* pICamera;
BSTR Vendor;
BYTE NumOfCameras;
HRESULT hr = pIManager->GetNumOfConnectedCameras(&NumOfCameras);
for (BYTE i = 0; i < NumOfCameras; ++i)
{
    hr = pIManager->GetCameraFromIndex(i, &pICamera);
    if (SUCCEEDED(hr))
    {
        pICamera->get_Vendor(&Vendor);
        if (CString(_T("Unibrain")) == CString(Vendor))
            break;
    }
}
```

## GetCameraFromGUID method

### Prototype

```
FireiXCamera GetCameraFromGUID(IN FireiGUID GUID)
```

### Comments

This method constructs a `FireiXCamera` object and returns it, given a valid `FireiGUID` object. The GUID represented by this `FireiGUID` must belong to a camera connected on the system, otherwise an error will occur.

It is most useful when the camera GUID is known beforehand, and provides a more direct way of accessing the camera as opposed to `GetCameraFromIndex`, because the positioning of the camera on the bus is irrelevant.

It is also mated to the SelectCamera method, as the FireiGUID returned by that method can be passed directly in GetCameraFromGUID to create the selected camera.

#### *Visual Basic 6.0 syntax*

```
Dim Camera As FireiXCamera
Dim GUID As FireiGUID
If Manager.SelectCamera(GUID) = True Then
    Set Camera = Manager.GetCameraFromGUID(GUID)
End If
```

#### *C++ syntax*

```
IFireiXCamera* pICamera;
IFireiGUID* pIGUID;
VARIANT_BOOL bSelected;
HRESULT hr = pIManager->SelectCamera(&pIGUID, &bSelected);
if (SUCCEEDED(hr) && bSelected)
    hr = pIManager->GetCameraFromGUID(pIGUID, &pICamera);
```

## **UseDirectShow method**

#### *Prototype*

```
UseDirectShow()
```

#### *Comments*

This method switches from Firei.dll API (FireAPI-based) to DirectShow API (FireiAPI-based) internal mode. The differences between the two modes are explained in greater detail in other areas of this text.

The call to UseDirectShow must be done as the first call immediately after constructing the FireiXManager object. Calling a different method or property first and then UseDirectShow will result in an error. Also, only a single call to UseDirectShow is possible in a single program – switching back and forth at runtime is not supported.

#### *Visual Basic 6.0 syntax*

```
Manager.UseDirectShow
```

#### *C++ syntax*

```
HRESULT hr = pIManager->UseDirectShow();
```

## ***FireiXRegister***

The `FireiXRegister` object is a helper object designed to make reading and writing values contained in a 4-byte register easy. It can be used to read/write any single bit of the register (represented as a Boolean value), or read/write any field of the register (represented as a Long value, and defined as from one bit to another bit).

It can be used in conjunction with the `Register` and `CommandRegister` properties of the camera, as those properties are represented by `FireiXRegister` objects.

### **Bit property**

#### ***Prototype***

```
Boolean Bit(IN Byte Index)
```

#### ***Comments***

The `Bit` property is used for reading or setting a single bit of the register, and is represented by a Boolean value; it is `True` when the bit is 1 and `False` when the bit is 0 in the `FireiXRegister` object.

The `Index` parameter can carry any value from 0 to 31, with 0 being the Most Significant Bit (represented in little-endian form, the same way cameras store their register values in them). If a value greater than 31 is passed, an error will occur.

It is the equivalent of using the `Field` property with the `FromIndex` and `ToIndex` parameters being equal to `Index`.

#### ***Visual Basic 6.0 syntax***

```
Dim Bit5 As Boolean
Bit5 = Reg.Bit(5)           'get
Reg.Bit(5) = Bit5          'set
```

#### ***C++ syntax***

```
VARIANT_BOOL bBit5;
HRESULT hr = pIReg->get_Bit(5, &bBit5); //get
hr = pIReg->put_Bit(5, bBit5);         //set
```

### **Field property**

#### ***Prototype***

```
Long Field(IN Byte FromIndex, IN Byte ToIndex)
```

#### ***Comments***

The `Field` property can be used to read or set a sequence of bits in the register, from a given bit up to and including another bit. It is represented by a Long value, so it can store the entire register as a number if desired.

The two `Index` parameters must both be from 0 to 31, and the `ToIndex` value must be greater than or equal to the `FromIndex` value. In any other case, an error occurs. Also, the Long value passed when setting the `Field` must be less than or equal to the maximum value that can be stored in the number of bits comprising

the field. So for instance, the examples below, which define a field of length of 3 bits can store a value from 0 to 7. If a value of 8 or greater was tried, an error would occur.

If FromIndex and ToIndex are equal, it is the same as calling the Bit property with the same value (albeit passing 0 or 1 as Long, instead of False or True as Boolean).

#### *Visual Basic 6.0 syntax*

```
Dim Value As Long
Value = Reg.Field(5, 7)           'get
Reg.Field(5, 7) = 6              'set
```

#### *C++ syntax*

```
LONG lValue;
HRESULT hr = pIReg->get_Field(5, 7, &lValue); //get
hr = pIReg->put_Field(5, 7, 6);             //set
```

## **FieldLen property**

#### *Prototype*

```
Long FieldLen(IN Byte FromIndex, IN Byte FieldLength)
```

#### *Comments*

#### *Visual Basic 6.0 syntax*

```
Dim Value As Long
Value = Reg.FieldLen(5, 3)       'get
Reg.FieldLen(5, 3) = 6          'set
```

#### *C++ syntax*

```
LONG lValue;
HRESULT hr = pIReg->get_FieldLen(5, 3, &lValue); //get
hr = pIReg->put_FieldLen(5, 3, 6);             //set
```

## **SwapEndian method**

#### *Prototype*

```
SwapEndian()
```

#### *Comments*

This method does a 32-bit endianness swap in the FireiXRegister object internally.

Please note that the entire value is swapped, meaning that any subsequent call to the FireiXRegister properties will yield different results. Also, two subsequent calls to SwapEndian have no effect on the value.

#### *Visual Basic 6.0 syntax*

```
Reg.SwapEndian
```

#### *C++ syntax*

```
HRESULT hr = pIReg->SwapEndian();
```

## ***FireIXTrigger***

The FireIXTrigger object encapsulates all the Trigger functionality that the camera may support. An instance cannot be constructed directly; it is rather accessed through the Trigger property of the camera.

Please note that the following properties have no additional functionality hidden inside them, besides what the camera itself supports. In essence, the entire FireIXTrigger object could have been two simple FireIXRegister objects (one for inquiry, one for control); it exists merely as a helping hand, in effect “naming” the bits and fields of the two registers as they map to the Trigger inquiry and control registers of the camera.

In order to call any of the following properties and methods, the IsSupported property must be True, otherwise each call will produce an error. Additionally, in order to read the value of any of the following properties, the CanRead property must also be True.

### **AbsControl property**

#### ***Prototype***

Boolean AbsControl

#### ***Comments***

This property reads/sets the Abs\_Control field of the camera trigger control register (bit 1). In order to access this property, the property HasAbsolute of the FireIXTrigger object must be True, otherwise an error will occur.

#### ***Visual Basic 6.0 syntax***

```
Dim Abs As Boolean
Abs = Trigger.AbsControl           'get
Trigger.AbsControl = True        'set
```

#### ***C++ syntax***

```
VARIANT_BOOL bAbs;
HRESULT hr = pITrigger->get_AbsControl(&Abs); //get
hr = pITrigger ->put_AbsControl(bAbs); //set
```

### **Enabled property**

#### ***Prototype***

Boolean Enabled

#### ***Comments***

This property reads/sets the ON\_OFF field of the camera trigger control register (bit 6). In order to access this property, the property HasOnOff of the FireIXTrigger object must be True, otherwise an error will occur.

#### ***Visual Basic 6.0 syntax***

```
Dim On As Boolean
On = Trigger.Enabled           'get
Trigger.Enabled = True        'set
```

### *C++ syntax*

```
VARIANT_BOOL bOn;  
HRESULT hr = pITrigger->get_Enabled(&bOn);           //get  
hr = pITrigger ->put_Enabled(bOn);                 //set
```

## **Polarity property**

### *Prototype*

Boolean Polarity

### *Comments*

This property reads/sets the Trigger\_Polarity field of the camera trigger control register (bit 7). In order to access this property, the property HasPolarity of the FireiXTrigger object must be True, otherwise an error will occur.

### *Visual Basic 6.0 syntax*

```
Dim Polarity As Boolean  
Polarity = Trigger.Polarity           'get  
Trigger.Polarity = True              'set
```

### *C++ syntax*

```
VARIANT_BOOL bPolarity;  
HRESULT hr = pITrigger->get_Polarity(&bPolarity); //get  
hr = pITrigger ->put_Polarity(bPolarity);       //set
```

## **Source property**

### *Prototype*

FireiXTriggerSource Source

### *Comments*

This property reads/sets the Trigger\_Source field of the camera trigger control register (bits 8 to 10). In order to set this property, a call to method IsSourceSupported of the FireiXTrigger object should first be performed, to check if the value being set is supported in this camera.

FireiXTriggerSource is an enumerated value, with possible values being tsSource\_0, tsSource\_1, tsSource\_2, tsSource\_3 and tsSource\_SW.

### *Visual Basic 6.0 syntax*

```
Dim TriggerSource As FireiXTriggerSource  
TriggerSource = Trigger.Source           'get  
Trigger.Source = tsSource_0             'set
```

### *C++ syntax*

```
FireiXTriggerSource Source;  
HRESULT hr = pITrigger->get_Source(&Source); //get  
hr = pITrigger ->put_Source(tsSource_0);   //set
```

## Value property

### Prototype

Boolean Value

### Comments

This is a read only property that reflects the `Trigger_Value` field of the camera trigger control register (bit 11). In order to read this property, the property `CanReadRaw` of the `FireiXTrigger` object must be `True`, otherwise an error will occur.

The resulting value can be considered as `False` being low and `True` being high signal value.

### Visual Basic 6.0 syntax

```
Dim Value As Boolean
Value = Trigger.Value
```

### C++ syntax

```
VARIANT_BOOL bValue;
HRESULT hr = pITrigger->get_Value(&bValue);
```

## Mode property

### Prototype

FireiXTriggerMode Mode

### Comments

This property reads/sets the `Trigger_Mode` field of the camera trigger control register (bits 12 to 15). In order to set this property, a call to method `IsModeSupported` of the `FireiXTrigger` object should first be performed, to check if the value being set is supported in this camera.

`FireiXTriggerMode` is an enumerated value, with possible values being `tmMode_0`, `tmMode_1`, `tmMode_2`, `tmMode_3`, `tmMode_4`, `tmMode_5`, `tmMode_14` and `tmMode_15`.

### Visual Basic 6.0 syntax

```
Dim TriggerMode As FireiXTriggerMode
TriggerMode = Trigger.Mode 'get
Trigger.Mode = tmMode_0 'set
```

### C++ syntax

```
FireiXTriggerMode Mode;
HRESULT hr = pITrigger->get_Mode(&Mode); //get
hr = pITrigger ->put_Mode(tmMode_0); //set
```

## IsSupported property

### Prototype

```
Boolean IsSupported
```

### Comments

This is a read only property that reflects the Presence\_Inq field of the camera trigger inquiry register (bit 0). This property can always be read, and it is a necessary requirement for it to be True for all the other methods and properties to have their intended behavior.

### Visual Basic 6.0 syntax

```
Dim Supported As Boolean  
Supported = Trigger.IsSupported
```

### C++ syntax

```
VARIANT_BOOL bSupported;  
HRESULT hr = pITrigger->get_IsSupported(&bSupported);
```

## HasAbsolute property

### Prototype

```
Boolean HasAbsolute
```

### Comments

This is a read only property that reflects the Abs\_Control\_Inq field of the camera trigger inquiry register (bit 1). It is a necessary requirement for it to be True for the AbsControl property to be accessible.

### Visual Basic 6.0 syntax

```
Dim HasAbs As Boolean  
HasAbs = Trigger.HasAbsolute
```

### C++ syntax

```
VARIANT_BOOL bHasAbs;  
HRESULT hr = pITrigger->get_HasAbsolute(&bHasAbs);
```

## CanRead property

### Prototype

```
Boolean CanRead
```

### Comments

This is a read only property that reflects the ReadOut\_Inq field of the camera trigger inquiry register (bit 4). It is a necessary requirement for it to be True for any of the control properties to be readable.

### Visual Basic 6.0 syntax

```
Dim CanRead As Boolean  
CanRead = Trigger.CanRead
```

### C++ syntax

```
VARIANT_BOOL bCanRead;  
HRESULT hr = pITrigger->get_CanRead(&bCanRead);
```

## HasOnOff property

### Prototype

```
Boolean HasOnOff
```

### Comments

This is a read only property that reflects the On/Off\_Inq field of the camera trigger inquiry register (bit 5). It is a necessary requirement for it to be True for the Enable property to be accessible.

### Visual Basic 6.0 syntax

```
Dim OnOff As Boolean  
OnOff = Trigger.HasOnOff
```

### C++ syntax

```
VARIANT_BOOL bOnOff;  
HRESULT hr = pITrigger->get_HasOnOff(&bOnOff);
```

## HasPolarity property

### Prototype

```
Boolean HasPolarity
```

### Comments

This is a read only property that reflects the Polarity\_Inq field of the camera trigger inquiry register (bit 6). It is a necessary requirement for it to be True for the Polarity property to be accessible.

### Visual Basic 6.0 syntax

```
Dim HasPol As Boolean  
HasPol = Trigger.HasPolarity
```

### C++ syntax

```
VARIANT_BOOL bHasPol;  
HRESULT hr = pITrigger->get_HasPolarity(&bHasPol);
```

## CanReadRaw property

### Prototype

```
Boolean CanReadRaw
```

### Comments

This is a read only property that reflects the Value\_Read\_Inq field of the camera trigger inquiry register (bit 7). It is a necessary requirement for it to be True for the Value property to be accessible.

### Visual Basic 6.0 syntax

```
Dim CanReadRaw As Boolean  
CanReadRaw = Trigger.CanReadRaw
```

### C++ syntax

```
VARIANT_BOOL bCanReadRaw;  
HRESULT hr = pITrigger->get_CanReadRaw(&bCanReadRaw);
```

## Parameter property

### Prototype

Integer Parameter

### Comments

This property reads/sets the Parameter field of the camera trigger control register (bits 20 to 31). Since this is an optional field for the trigger, there is no equivalent inquiry to be made whether it is available or not. However, acceptable values are integers greater than or equal to zero and less than 4096.

### Visual Basic 6.0 syntax

```
Dim Parameter As Integer
Parameter = Trigger.Parameter           'get
Trigger.Parameter = 1000               'set
```

### C++ syntax

```
SHORT ushParam;
HRESULT hr = pITrigger->get_Parameter(&ushParam); //get
hr = pITrigger->put_Parameter(ushParam);         //set
```

## IsSourceSupported method

### Prototype

Boolean IsSourceSupported(IN FireiXTriggerSource Source)

### Comments

This method checks whether a given FireiXTriggerSource value is supported by the camera. The programmer should typically test for the support of a trigger source value, before trying to set it using the Source property.

### Visual Basic 6.0 syntax

```
Dim Supported As Boolean
Supported = Trigger.IsSourceSupported(tsSource_0)
```

### C++ syntax

```
VARIANT_BOOL bSupported;
HRESULT hr =
    pITrigger->get_IsSourceSupported(tsSource_0, &bSupported);
```

## IsModeSupported method

### Prototype

Boolean IsModeSupported(IN FireiXTriggerMode Mode)

### Comments

This method checks whether a given FireiXTriggerMode value is supported by the camera. The programmer should typically test for the support of a trigger mode value, before trying to set it using the Mode property.

### Visual Basic 6.0 syntax

```
Dim Supported As Boolean
```

```
Supported = Trigger.IsModeSupported(tmMode_0)
```

#### *C++ syntax*

```
VARIANT_BOOL bSupported;  
HRESULT hr = pITrigger->get_IsModeSupported(tmMode_0, &bSupported);
```

## **Reload method**

#### *Prototype*

```
Reload()
```

#### *Comments*

This method reloads the values of the trigger control register from the camera, so effectively it acts as an “undo” function for any changes made through the various properties, as long as Save hasn’t yet been called.

The inquiry register on the other hand is not reloaded, since it contains read-only values which cannot be changed anyway.

#### *Visual Basic 6.0 syntax*

```
Trigger.Reload
```

#### *C++ syntax*

```
HRESULT hr = pITrigger->Reload();
```

## **PullSoftwareTrigger method**

#### *Prototype*

```
PullSoftwareTrigger()
```

#### *Comments*

This method sets the software\_trigger register of the camera for acquiring a frame. If the source property of the trigger is not set to Source\_SW, an exception will be thrown.

You can call PullSoftwareTrigger method subsequently for acquiring new frames.

#### *Visual Basic syntax*

```
Trigger.PullSoftwareTrigger
```

#### *C# syntax*

```
HRESULT hr = pITrigger->PullSoftwareTrigger();
```

## **Save method**

#### *Prototype*

```
Save()
```

#### *Comments*

This method saves any changes made to the trigger control register of the camera. If for some reason this cannot be completed (i.e., the camera refuses the write request), an error will occur.

Any subsequent calls to the Reload method will revert to this saved state.

#### *Visual Basic 6.0 syntax*

```
Trigger.Save
```

#### *C++ syntax*

```
HRESULT hr = pITrigger->Save();
```

### **FireiXStreamFormat**

The FireiXStreamFormat object encapsulates the video format settings relevant a given camera. A FireiXStreamFormat object instance cannot and should not be created directly; its functionality is paired closely with the specific camera, so the camera is responsible for constructing the object.

To obtain a FireiXStreamFormat, the programmer can use the StreamFormat property of FireiXCamera. Additionally, it can be obtained through the EnumFireiXStreamFormats enumerator; since the enumerator is created by the camera, all the FireiXStreamFormats obtained through it, are connected with that camera automatically. It can also be obtained directly through the two “retrieve” methods, RetrieveStreamFormat and RetrieveStreamFormatFromIdentifier.

The following properties and methods are not final; changes are kept in the FireiXStreamFormat object and only passed to the camera as the current settings when the Save method is called (or the FireiXStreamFormat is set to the camera using the StreamFormat property). They can also be reverted to the currently saved in the camera values using the Reload method.

### **PixelFormatString property**

#### *Prototype*

```
String PixelFormatString
```

#### *Comments*

This is a read only property that returns the pixel format of the FireiXStreamFormat as a string. A pixel format is not unique to a FireiXStreamFormat. A pair of a pixel format and a resolution however is.

#### *Visual Basic 6.0 syntax*

```
Dim PixelFormat As String  
PixelFormat = StreamFormat.PixelFormatString
```

#### *C++ syntax*

```
BSTR PixelFormat;  
HRESULT hr = pIStreamFormat->get_PixelFormatString(&PixelFormat);
```

### **PixelFormat property**

#### *Prototype*

```
FireiXPixelFormat PixelFormat
```

### *Comments*

This is a read only property that returns the pixel format of the `FireiXStreamFormat` as a `FireiXPixelFormat` value. A pixel format is not unique to a `FireiXStreamFormat`. A pair of a pixel format and a resolution however is.

Possible values in the `FireiXPixelFormat` enumerated value are:

```
pfNone,  
pfY_MONO,  
pfYUV_411,  
pfYUV_422,  
pfYUV_444,  
pfRGB_24,  
pfY_MONO_16,  
pfRGB_48,  
pfS_Y_MONO_16,  
pfS_RGB_48,  
pfRAW_8,  
pfRAW_16
```

### *Visual Basic 6.0 syntax*

```
Dim PixelFormat As FireiXPixelFormat  
PixelFormat = StreamFormat.PixelFormat
```

### *C++ syntax*

```
FireiXPixelFormat PixelFormat;  
HRESULT hr = pIStreamFormat->get_PixelFormat(&PixelFormat);
```

## **Resolution property**

### *Prototype*

```
FireiXResolution Resolution
```

### *Comments*

This is a read only property that returns the resolution of the `FireiXStreamFormat` as a `FireiXResolution` value. A resolution is not unique to a `FireiXStreamFormat`. A pair of a pixel format and a resolution however is.

Possible values in the `FireiXResolution` enumerated value are:

```
resNone,  
res160x120,  
res320x240,  
res640x480,  
res800x600,  
res1024x768,  
res1280x960,  
res1600x1200,  
resVariable,  
resVariable_1,  
resVariable_2,  
resVariable_3,
```

```
resVariable_4,  
resVariable_5,  
resVariable_6,  
resVariable_7
```

The values from res160x120 up to res1600x1200 represent Fixed formats, and from resVariable to resVariable\_7 represent User Defined formats.

#### *Visual Basic 6.0 syntax*

```
Dim Resolution As FireiXResolution  
Resolution = StreamFormat.Resolution
```

#### *C++ syntax*

```
FireiXResolution Resolution;  
HRESULT hr = pIStreamFormat->get_Resolution(&Resolution);
```

### **IsUserDefined property**

#### *Prototype*

```
Boolean IsUserDefined
```

#### *Comments*

This is a read only property that returns whether the given FireiXStreamFormat is User Defined. If True, the UserDefined property of FireiXStreamFormat can be used. Otherwise, the Fixed property can be used.

#### *Visual Basic 6.0 syntax*

```
Dim UserDefined As Boolean  
UserDefined = StreamFormat.IsUserDefined
```

#### *C++ syntax*

```
VARIANT_BOOL bUserDefined;  
HRESULT hr = pIStreamFormat->get_IsUserDefined(&bUserDefined);
```

### **IsCurrent property**

#### *Prototype*

```
Boolean IsCurrent
```

#### *Comments*

This is a read only property that returns whether the given FireiXStreamFormat is the currently selected streaming format of the camera. Please note that it does not necessarily mean that all parameters of the stream format are saved on the camera; it merely means it has the same pixel format and resolution.

If the FireiXStreamFormat was obtained through the StreamFormat property of FireiXCamera (which returns the currently selected FireiXStreamFormat) this property should always be True. It is therefore only useful if the FireiXStreamFormat object was obtained by other means.

#### *Visual Basic 6.0 syntax*

```
Dim Current As Boolean  
Current = StreamFormat.IsCurrent
```

### *C++ syntax*

```
VARIANT_BOOL bCurrent;  
HRESULT hr = pIStreamFormat->get_IsCurrent(&bCurrent);
```

## **Description property**

### *Prototype*

```
String Description
```

### *Comments*

This is a read only property that returns a textual representation of the `FireiXStreamFormat`. The text contains the pixel format and the resolution of the streaming format. A typical description for a Fixed format would look like:

```
Y_MONO, 160 X 120
```

In case of a User Defined format the string contains the maximum resolution instead, similar to:

```
Y_MONO, Max 160 X 120
```

### *Visual Basic 6.0 syntax*

```
Dim Description As String  
Description = StreamFormat.Description
```

### *C++ syntax*

```
BSTR Description;  
HRESULT hr = pIStreamFormat->get_Description(&Description);
```

## **Identifier property**

### *Prototype*

```
Long Identifier
```

### *Comments*

This is a read only property that returns a unique identifier, in the form of a Long value that can be stored for later usage (e.g., to retrieve this format directly from the camera using `RetrieveStreamFormatFromIdentifier`).

This identifier is ideal for storing the stream format selection to a windows control, such as a Combo Box or a List Box (as each entry's data). It should be noted however, that the variable settings of `FireiXStreamFormat` are not transferred through this unique identifier; only the constant attributes of a `FireiXStreamFormat`, i.e., the pixel format and resolution.

### *Visual Basic 6.0 syntax*

```
Dim ID As Long  
ID = StreamFormat.Identifier
```

### *C++ syntax*

```
LONG IID;  
HRESULT hr = pIStreamFormat->get_Identifier(&IID);
```

## Fixed property

### Prototype

```
FireiXFixedStreamFormat Fixed
```

### Comments

This is a read only property that returns the associated `FireiXFixedStreamFormat` object with this `FireiXStreamFormat`. It is only valid if the `IsUserDefined` property is `False` (an error would occur trying to access this property with `IsUserDefined` being `True`).

### Visual Basic 6.0 syntax

```
Dim Fixed As FireiXFixedStreamFormat  
Set Fixed = StreamFormat.Fixed
```

### C++ syntax

```
IFireiXFixedStreamFormat* pIFixed;  
HRESULT hr = pIStreamFormat->get_Fixed(&pIFixed);
```

## UserDefined property

### Prototype

```
FireiXUserDefinedStreamFormat UserDefined
```

### Comments

This is a read only property that returns the associated `FireiXUserDefinedStreamFormat` object with this `FireiXStreamFormat`. It is only valid if the `IsUserDefined` property is `True` (an error would occur trying to access this property with `IsUserDefined` being `False`).

### Visual Basic 6.0 syntax

```
Dim UserDefined As FireiXUserDefinedStreamFormat  
Set UserDefined = StreamFormat.UserDefined
```

### C++ syntax

```
IFireiXUserDefinedStreamFormat* pIUserDefined;  
HRESULT hr = pIStreamFormat->get_UserDefined(&pIUserDefined);
```

## RawModeOverride property

### Prototype

```
FireiXRawMode RawModeOverride
```

### Comments

This is a read/write property that allows the programmer to establish if a bayer conversion will be performed on the image data coming from the camera and if so, the color filter that the camera is using.

The default setting is `rmAuto`, which means the SDK will try to determine the correct conversion necessary. This is not always possible, since many raw-mode cameras sent raw-data as regular `Y_MONO`. In those cases, an override is necessary if a conversion to RGB is desired, hence this property.

Per the IIDC specification, only under User Defined modes (Format\_7) a camera is allowed to send raw unconverted data – in that case, the color filter is supplied by the camera. However, many camera manufacturers disguise raw data as Y\_MONO, sent either through Fixed or User Defined streaming formats.

The RawModeOverride property carries enumerated values; possibilities are:

```
rmAuto,  
rmNone,  
rmRGGb,  
rmGRbG,  
rmGBRg,  
rmBGgR
```

#### *Visual Basic 6.0 syntax*

```
Dim RawMode As FireiXRawMode  
RawMode = StreamFormat.RawModeOverride  
StreamFormat.RawModeOverride = rmRGGb 'get  
                                     'set
```

#### *C++ syntax*

```
FireiXRawMode RawMode;  
HRESULT hr = pIStreamFormat->get_RawModeOverride(&RawMode); //get  
hr = pIStreamFormat->put_RawModeOverride(rmRGGb); //set
```

## **Width property**

### *Prototype*

Integer Width

### *Comments*

This is a read only property that returns the resolution width of the streaming format. This is constant for a Fixed streaming format and variable on a User Defined.

### *Visual Basic 6.0 syntax*

```
Dim Width As Integer  
Width = StreamFormat.Width
```

### *C++ syntax*

```
SHORT shWidth;  
HRESULT hr = pIStreamFormat->get_Width(&shWidth);
```

## **Height property**

### *Prototype*

Integer Height

### *Comments*

This is a read only property that returns the resolution height of the streaming format. This is constant for a Fixed streaming format and variable on a User Defined.

### *Visual Basic 6.0 syntax*

```
Dim Height As Integer
```

```
Height = StreamFormat.Height
```

### *C++ syntax*

```
SHORT shHeight;  
HRESULT hr = pIStreamFormat->get_Height(&shHeight);
```

## **Save method**

### *Prototype*

```
Save()
```

### *Comments*

This method sets this `FireiXStreamFormat` to the camera, including any selected attributes, whether Fixed or User Defined. If for some reason this cannot be completed (i.e., the camera refuses the write request), an error will occur.

Any subsequent calls to the `Reload` method will revert to this saved state.

The effect of this method is the same as using the `StreamFormat` property of `FireiXCamera`, to set this `FireiXStreamFormat`.

### *Visual Basic 6.0 syntax*

```
StreamFormat.Save
```

### *C++ syntax*

```
HRESULT hr = pIStreamFormat->Save();
```

## **Reload method**

### *Prototype*

```
Reload()
```

### *Comments*

This method reloads the default values of this streaming format from the camera, so effectively it acts as an “undo” function for any changes made through the various properties, as long as `Save` hasn’t yet been called.

### *Visual Basic 6.0 syntax*

```
StreamFormat.Reload
```

### *C++ syntax*

```
HRESULT hr = pIStreamFormat->Reload();
```

## ***FireiXFixedStreamFormat***

The `FireiXFixedStreamFormat` object is derived from `FireiXStreamFormat`. It carries any additional functionality that pertains to Fixed streaming formats. It cannot be constructed directly; it is only ever obtained through the `Fixed` property of `FireiXStreamFormat`.

Its properties are mostly read-only, except for the `FrameRate`, which can be set by the programmer.

### **ResolutionString property**

#### *Prototype*

```
String ResolutionString
```

#### *Comments*

This is a read only property that returns the resolution of the `FireiXFixedStreamFormat` as a string. A resolution is not unique to a `FireiXStreamFormat`. A pair of a pixel format and a resolution however is.

#### *Visual Basic 6.0 syntax*

```
Dim Resolution As String  
Resolution = Fixed.ResolutionString
```

#### *C++ syntax*

```
BSTR Resolution;  
HRESULT hr = pIFixed->get_ResolutionString(&Resolution);
```

### **FrameRateString property**

#### *Prototype*

```
String FrameRateString
```

#### *Comments*

This is a read/write property that can be used to retrieve a textual representation of the frame rate of the streaming format and then set it back.

Since the format of the text is specific, setting the property can be safely done only with values got from it.

#### *Visual Basic 6.0 syntax*

```
Dim FrameRate As String  
FrameRate = Fixed.FrameRateString  
Fixed.FrameRateString = FrameRate
```

```
'get  
'set
```

#### *C++ syntax*

```
BSTR FrameRate;  
HRESULT hr = pIFixed->get_FrameRateString(&FrameRate);  
hr = pIFixed->put_FrameRateString(FrameRate);
```

```
//get  
//set
```

### **FrameRate property**

#### *Prototype*

```
FireiXFrameRate FrameRate
```

### Comments

This is a read/write property that can be used to retrieve or set the frame rate of the `FireiXFixedStreamFormat`. It is represented by an enumerated value (`FireiXFrameRate`), with the following possibilities:

```
fpsNone,  
fps1_875,  
fps3_75,  
fps7_5,  
fps15,  
fps30,  
fps60,  
fps120,  
fps240
```

The programmer can test whether a given frame rate is valid with this Fixed format and acceptable by the camera, a call to `IsFrameRateSupported` is sufficient.

### Visual Basic 6.0 syntax

```
Dim FrameRate As FireiXFrameRate  
FrameRate = Fixed.FrameRate  
Fixed.FrameRate = fps30
```

`'get`  
`'set`

### C++ syntax

```
FireiXFrameRate FrameRate;  
HRESULT hr = pIFixed->get_FrameRate(&FrameRate);  
hr = pIFixed->put_FrameRate(FrameRate);
```

`//get`  
`//set`

## Width property

### Prototype

Integer Width

### Comments

This is a read only property that returns the resolution width of the streaming format. This will be constant between successive reads, since the width is always the same on a given Fixed streaming format.

### Visual Basic 6.0 syntax

```
Dim Width As Integer  
Width = Fixed.Width
```

### C++ syntax

```
SHORT shWidth;  
HRESULT hr = pIFixed->get_Width(&shWidth);
```

## Height property

### Prototype

Integer Height

### *Comments*

This is a read only property that returns the resolution height of the streaming format. This will be constant between successive reads, since the height is always the same on a given Fixed streaming format.

### *Visual Basic 6.0 syntax*

```
Dim Height As Integer  
Height = Fixed.Height
```

### *C++ syntax*

```
SHORT shHeight;  
HRESULT hr = pIFixed->get_Height(&shHeight);
```

## **PacketSize property**

### *Prototype*

```
Long PacketSize
```

### *Comments*

This is a read only property that returns the size in bytes of the packets for this streaming format. This will be constant between successive reads, since the packet size is always the same on a given Fixed streaming format.

### *Visual Basic 6.0 syntax*

```
Dim PacketSize As Long  
PacketSize = Fixed.PacketSize
```

### *C++ syntax*

```
LONG lPacketSize;  
HRESULT hr = pIFixed->get_PacketSize(&lPacketSize);
```

## **PacketsPerFrame property**

### *Prototype*

```
Long PacketsPerFrame
```

### *Comments*

This is a read only property that returns the number of packets per frame for this streaming format. This will be constant between successive reads, since the number of packets per frame is always the same on a given Fixed streaming format.

### *Visual Basic 6.0 syntax*

```
Dim PacketsPerFrame As Long  
PacketsPerFrame = Fixed.PacketsPerFrame
```

### *C++ syntax*

```
LONG lPacketsPerFrame;  
HRESULT hr = pIFixed->get_PacketsPerFrame(&lPacketsPerFrame);
```

## Description property

### Prototype

String Description

### Comments

This is a read only property that returns a textual representation of the FireiXFixedStreamFormat. The text contains the pixel format and the resolution of the streaming format. A typical description would look like:

```
Y_MONO, 160 X 120
```

It is also the same value returned by the Description property of the FireiXStreamFormat from which this FireiXFixedStreamFormat was derived from.

### Visual Basic 6.0 syntax

```
Dim Description As String  
Description = Fixed.Description
```

### C++ syntax

```
BSTR Description;  
HRESULT hr = pIFixed->get_Description(&Description);
```

## IsFrameRateSupported method

### Prototype

```
Boolean IsFrameRateSupported(IN FireiXFrameRate FrameRate)
```

### Comments

This method will return True if the supplied FireiXFrameRate is both available in this FireiXFixedStreamFormat and acceptable by the camera and the current conditions of the 1394 bus.

A frame rate may not be available on a given Fixed streaming format, as defined in the IIDC spec. It may also not be supported by the camera; these are static conditions. There are dynamic conditions though, such as the isochronous speed of the bus the camera is connected to, limited by both hardware and software. Some frame rates may require a higher isochronous speed than what is available currently, and even though the camera may support the frame rate, it might not be currently achievable. In all those cases, IsFrameRateSupported will return False.

### Visual Basic 6.0 syntax

```
Dim Supported As Boolean  
Supported = Fixed.IsFrameRateSupported fps30
```

### C++ syntax

```
VARIANT_BOOL bSupported;  
HRESULT hr = pIFixed->IsFrameRateSupported(fps30, &bSupported);
```

## Save method

### *Prototype*

```
Save()
```

### *Comments*

This method sets this `FireiXFixedStreamFormat` to the camera, including the selected frame rate. If for some reason this cannot be completed (i.e., the camera refuses the write request), an error will occur.

Any subsequent calls to the `Reload` method will revert to this saved state.

The effect of this method is the same as using the `StreamFormat` property of `FireiXCamera`, to set the `FireiXStreamFormat` from which this `FireiXFixedStreamFormat` was derived.

### *Visual Basic 6.0 syntax*

```
Fixed.Save
```

### *C++ syntax*

```
HRESULT hr = pIFixed->Save();
```

## Reload method

### *Prototype*

```
Reload()
```

### *Comments*

This method reloads the default values of this streaming format from the camera, so effectively it acts as an “undo” function for any changes made in the frame rate, as long as `Save` hasn’t yet been called.

### *Visual Basic 6.0 syntax*

```
Fixed.Reload
```

### *C++ syntax*

```
HRESULT hr = pIFixed->Reload();
```

## ***FireiXUserDefinedStreamFormat***

The `FireiXUserDefinedStreamFormat` object is derived from `FireiXStreamFormat`. It carries any additional functionality that pertains to User Defined streaming formats. It cannot be constructed directly; it is only ever obtained through the `UserDefined` property of `FireiXStreamFormat`.

Unlike the `FireiXFixedStreamFormat` object, there many selectable attributes in `FireiXUserDefinedStreamFormat`.

### **Left property**

#### *Prototype*

```
Integer Left
```

#### *Comments*

This is a read only property that can be used to read the currently set left coordinate of the Region of Interest set in the `FireiXUserDefinedStreamFormat`. This can be set through the `SetROI` method.

#### *Visual Basic 6.0 syntax*

```
Dim Left As Integer  
Left = UserDefined.Left
```

#### *C++ syntax*

```
SHORT shLeft;  
HRESULT hr = pIUserDefined->get_Left(&shLeft);
```

### **Right property**

#### *Prototype*

```
Integer Right
```

#### *Comments*

This is a read only property that can be used to read the currently set right coordinate of the Region of Interest set in the `FireiXUserDefinedStreamFormat`. This can be set through the `SetROI` method.

#### *Visual Basic 6.0 syntax*

```
Dim Right As Integer  
Right = UserDefined.Right
```

#### *C++ syntax*

```
SHORT shRight;  
HRESULT hr = pIUserDefined->get_Right(&shRight);
```

### **Top property**

#### *Prototype*

```
Integer Top
```

### *Comments*

This is a read only property that can be used to read the currently set top coordinate of the Region of Interest set in the FireiXUserDefinedStreamFormat. This can be set through the SetROI method.

### *Visual Basic 6.0 syntax*

```
Dim Top As Integer  
Top = UserDefined.Top
```

### *C++ syntax*

```
SHORT shTop;  
HRESULT hr = pIUserDefined->get_Top(&shTop);
```

## **Bottom property**

### *Prototype*

```
Integer Bottom
```

### *Comments*

This is a read only property that can be used to read the currently set bottom coordinate of the Region of Interest set in the FireiXUserDefinedStreamFormat. This can be set through the SetROI method.

### *Visual Basic 6.0 syntax*

```
Dim Bottom As Integer  
Bottom = UserDefined.Bottom
```

### *C++ syntax*

```
SHORT shBottom;  
HRESULT hr = pIUserDefined->get_Bottom(&shBottom);
```

## **Width property**

### *Prototype*

```
Integer Width
```

### *Comments*

This is a read only property that can be used to read the currently set width of the Region of Interest set in the FireiXUserDefinedStreamFormat. This can be set through the SetROI method and it is equal to  $Right - Left$ . It is also the same value returned by the Width property of the FireiXStreamFormat from which this FireiXUserDefinedStreamFormat was derived from.

### *Visual Basic 6.0 syntax*

```
Dim Width As Integer  
Width = UserDefined.Width
```

### *C++ syntax*

```
SHORT shWidth;  
HRESULT hr = pIUserDefined->get_Width(&shWidth);
```

## Height property

### Prototype

Integer Height

### Comments

This is a read only property that can be used to read the currently set height of the Region of Interest set in the FireiXUserDefinedStreamFormat. This can be set through the SetROI method and it is equal to Bottom - Top. It is also the same value returned by the Height property of the FireiXStreamFormat from which this FireiXUserDefinedStreamFormat was derived from.

### Visual Basic 6.0 syntax

```
Dim Height As Integer  
Height = UserDefined.Height
```

### C++ syntax

```
SHORT shHeight;  
HRESULT hr = pIUserDefined->get_Height(&shHeight);
```

## MaxWidth property

### Prototype

Integer MaxWidth

### Comments

This is a read only property that can be used to read the maximum possible width for the Region of Interest of the FireiXUserDefinedStreamFormat.

### Visual Basic 6.0 syntax

```
Dim MaxWidth As Integer  
MaxWidth = UserDefined.MaxWidth
```

### C++ syntax

```
SHORT shMaxWidth;  
HRESULT hr = pIUserDefined->get_MaxWidth(&shMaxWidth);
```

## MaxHeight property

### Prototype

Integer MaxHeight

### Comments

This is a read only property that can be used to read the maximum possible height for the Region of Interest of the FireiXUserDefinedStreamFormat.

### Visual Basic 6.0 syntax

```
Dim MaxHeight As Integer  
MaxHeight = UserDefined.MaxHeight
```

### C++ syntax

```
SHORT shMaxHeight;
```

```
HRESULT hr = pIUserDefined->get_MaxHeight(&shMaxHeight);
```

## HorizontalPositionUnit property

### Prototype

```
Integer HorizontalPositionUnit
```

### Comments

This is a read only property that can be used to read the horizontal unit for the position of the Region of Interest of the FireiXUserDefinedStreamFormat. The left, right, top and bottom x-coordinates of the ROI must be perfectly divisible by this value.

### Visual Basic 6.0 syntax

```
Dim HPosUnit As Integer  
HPosUnit = UserDefined.HorizontalPositionUnit
```

### C++ syntax

```
SHORT shHPosUnit;  
HRESULT hr = pIUserDefined->get_HorizontalPositionUnit(&shHPosUnit);
```

## VerticalPositionUnit property

### Prototype

```
Integer VerticalPositionUnit
```

### Comments

This is a read only property that can be used to read the vertical unit for the position of the Region of Interest of the FireiXUserDefinedStreamFormat. The left, right, top and bottom y-coordinates of the ROI must be perfectly divisible by this value.

### Visual Basic 6.0 syntax

```
Dim VPosUnit As Integer  
VPosUnit = UserDefined.VerticalPositionUnit
```

### C++ syntax

```
SHORT shVPosUnit;  
HRESULT hr = pIUserDefined->get_VerticalPositionUnit(&shVPosUnit);
```

## WidthUnit property

### Prototype

```
Integer WidthUnit
```

### Comments

This is a read only property that can be used to read the width unit for the Region of Interest of the FireiXUserDefinedStreamFormat. The total width of the ROI must be perfectly divisible by this value.

### Visual Basic 6.0 syntax

```
Dim WUnit As Integer  
WUnit = UserDefined.WidthUnit
```

### *C++ syntax*

```
SHORT shWUnit;  
HRESULT hr = pIUserDefined->get_WidthUnit(&shWUnit);
```

## **HeightUnit property**

### *Prototype*

```
Integer HeightUnit
```

### *Comments*

This is a read only property that can be used to read the height unit for the Region of Interest of the FireiXUserDefinedStreamFormat. The total height of the ROI must be perfectly divisible by this value.

### *Visual Basic 6.0 syntax*

```
Dim HUnit As Integer  
HUnit = UserDefined.HeightUnit
```

### *C++ syntax*

```
SHORT shHUnit;  
HRESULT hr = pIUserDefined->get_HeightUnit(&shHUnit);
```

## **MaxPacketSize property**

### *Prototype*

```
Long MaxPacketSize
```

### *Comments*

This is a read only property that can be used to read the maximum allowed packet size in bytes of the FireiXUserDefinedStreamFormat. The PacketSize property cannot be set to a value greater than this value.

### *Visual Basic 6.0 syntax*

```
Dim MaxPacketSize As Long  
MaxPacketSize = UserDefined.MaxPacketSize
```

### *C++ syntax*

```
LONG lMaxPacketSize;  
HRESULT hr = pIUserDefined->get_MaxPacketSize(&lMaxPacketSize);
```

## **PacketSizeUnit property**

### *Prototype*

```
Long PacketSizeUnit
```

### *Comments*

This is a read only property that can be used to read the packet size unit of the FireiXUserDefinedStreamFormat. The PacketSize property must be set to a value perfectly divisibly by this value.

### *Visual Basic 6.0 syntax*

```
Dim PacketSizeUnit As Long
```

```
PacketSizeUnit = UserDefined.PacketSizeUnit
```

### *C++ syntax*

```
LONG lPacketSizeUnit;  
HRESULT hr = pIUserDefined->get_PacketSizeUnit(&lPacketSizeUnit);
```

## **PacketSize property**

### *Prototype*

```
Long PacketSize
```

### *Comments*

This is a read/write property that can be used to retrieve or set the size in bytes of each packet the camera will transmit for this format.

When setting it, acceptable values are greater than or equal to 0 and less than or equal to the MaxPacketSize property. The value must also be perfectly divisible by the value of PacketSizeUnit property.

Setting this value to 0 means the SDK will try and determine the correct packet size automatically, either by setting it to the maximum possible, or by querying the camera for it.

### *Visual Basic 6.0 syntax*

```
Dim PacketSize As Long  
PacketSize = UserDefined.PacketSize  
UserDefined.PacketSize = 0
```

*'get  
'set*

### *C++ syntax*

```
LONG lPacketSize;  
HRESULT hr = pIUserDefined->get_PacketSize(&lPacketSize);  
hr = pIUserDefined->put_PacketSize(0);
```

*//get  
//set*

## **Description property**

### *Prototype*

```
String Description
```

### *Comments*

This is a read only property that returns a textual representation of the FireiXUserDefinedStreamFormat. The text contains the pixel format and the maximum resolution of the streaming format. A typical description would look like:

```
Y_MONO, Max 160 X 120
```

It is also the same value returned by the Description property of the FireiXStreamFormat from which this FireiXUserDefinedStreamFormat was derived from.

### *Visual Basic 6.0 syntax*

```
Dim Description As String  
Description = UserDefined.Description
```

### *C++ syntax*

```
BSTR Description;  
HRESULT hr = pIUserDefined->get_Description(&Description);
```

## **SetROI method**

### *Prototype*

```
Boolean SetROI(  
    IN Integer Left,  
    IN Integer Top,  
    IN Integer Width  
    IN Integer Height  
)
```

### *Comments*

This method is used to set the desired coordinates and size of the Region of Interest for the FireiXUserDefinedStreamFormat.

The Left and Top values must be greater than or equal to 0 and less than or equal to the MaxWidth and MaxHeight properties respectively. They must also be perfectly divisible by HorizontalPositionUnit and VerticalPositionUnit respectively.

The Width and Height values must be greater than or equal to WidthUnit and HeightUnit respectively and less than or equal to MaxWidth and MaxHeight respectively. They must also be perfectly divisible by WidthUnit and HeightUnit respectively. Additionally, they must not be greater than the available rectangle as set by the Left and Top values (i.e., the sum of Width and Left must not be greater than MaxWidth and the sum of Height and Top must not be greater than MaxHeight).

The method will return True or False, depending on whether the above conditions are met. It will not query the camera for validity; any checks are made inside the SDK, according to the reported parameters of the camera.

### *Visual Basic 6.0 syntax*

```
Dim Result As Boolean  
Result = UserDefined.SetROI(0, 0, 160, 120)
```

### *C++ syntax*

```
VARIANT_BOOL bResult;  
HRESULT hr = pIUserDefined->SetROI(0, 0, 160, 120, &bResult);
```

## **IsValid method**

### *Prototype*

```
Boolean IsValid()
```

### *Comments*

This method will return True if the currently set properties of the FireiXUserDefinedStreamFormat are acceptable by the camera.

This is useful in those cases where the camera reported acceptable values are not 100% correct. For instance, even though theoretically a camera should support a ROI width from `WidthUnit` to `MaxWidth`, this is not always the case in practice. Since calling `Save` produces an error in that case, calling `IsValid` beforehand can provide a measure of safety.

#### *Visual Basic 6.0 syntax*

```
Dim Valid As Boolean
Valid = UserDefined.IsValid
```

#### *C++ syntax*

```
VARIANT_BOOL bValid;
HRESULT hr = pIUserDefined->IsValid(&bValid);
```

## **Save method**

#### *Prototype*

```
Save()
```

#### *Comments*

This method sets this `FireiXUserDefinedStreamFormat` to the camera, including the selected Region of Interest and `PacketSize`. If for some reason this cannot be completed (i.e., the camera refuses the write request), an error will occur. To prevent this, a call to `IsValid` first can be performed.

Any subsequent calls to the `Reload` method will revert to this saved state.

The effect of this method is the same as using the `StreamFormat` property of `FireiXCamera`, to set the `FireiXStreamFormat` from which this `FireiXUserDefinedStreamFormat` was derived.

#### *Visual Basic 6.0 syntax*

```
UserDefined.Save
```

#### *C++ syntax*

```
HRESULT hr = pIUserDefined->Save();
```

## **Reload method**

#### *Prototype*

```
Reload()
```

#### *Comments*

This method reloads the default values of this streaming format from the camera, so effectively it acts as an “undo” function for any changes made in the Region of Interest and `PacketSize`, as long as `Save` hasn’t yet been called.

#### *Visual Basic 6.0 syntax*

```
UserDefined.Reload
```

#### *C++ syntax*

```
HRESULT hr = pIUserDefined->Reload();
```

## ***EnumFireiXStreamFormats***

The EnumFireiXStreamFormats enumerator can be used to iterate between a set of FireiXStreamFormat objects. It cannot be constructed directly, it is obtained through the GetStreamFormatsEnumerator method of FireiXCamera.

As is the case with EnumFireiXFeatures, this object implements only two methods that are sufficient for iterating.

### **Reset method**

#### *Prototype*

```
Reset()
```

#### *Comments*

This method will start the iteration from the beginning, i.e., the next call to Next will return the first FireiXStreamFormat.

It is not necessary to call this method on a newly constructed EnumFireiXStreamFormats object, it will start from the beginning by default.

#### *Visual Basic 6.0 syntax*

```
StreamFormats.Reset
```

#### *C++ syntax*

```
HRESULT hr = pIStreamFormats->Reset();
```

### **Next method**

#### *Prototype*

```
Boolean Next(OUT FireiXStreamFormat StreamFormat)
```

#### *Comments*

This can be used to obtain the next FireiXStreamFormat in EnumFireiXStreamFormats. It will return True if another item is available, and False if this was the last item. To start again from the first item, a call to Reset is sufficient.

#### *Visual Basic 6.0 syntax*

```
Do While StreamFormats.Next(StreamFormat) = True  
    ...  
Loop
```

#### *C++ syntax*

```
VARIANT_BOOL bHasNext;  
HRESULT hr = pIStreamFormats->Next(&pIStreamFormat, &bHasNext);  
while (bHasNext)  
{  
    ...  
    hr = pIStreamFormats->Next(&pIStreamFormat, &bHasNext);  
}
```

## ***FireiXFeature***

The `FireiXFeature` object encapsulates all functionality related to a camera feature. It contains all the information available through the feature's inquiry register, and all the capabilities available through the feature's control register in one concise package.

The `FireiXFeature` object cannot be constructed directly by the programmer; instead, it is accessible through the camera properties at any time. The `FireiXCamera` object offers a specialized read-only property for accessing each individual feature. For example, it has a `Shutter` property that returns the `FireiXFeature` object pertaining to the shutter feature. If a more generic way to access features is required, the `Feature` property can be used instead; it takes a string of the name of the feature as a parameter.

Additionally, there is an `EnumFireiXFeatures` enumerated object, operating in much the same way as the `EnumFireiXStreamFormats` enumerator, and it allows iterating between the features of the camera. This is enumerator is accessible through the `GetFeaturesEnumerator` method of `FireiXCamera`.

The `FireiXFeature` object implements various properties and methods to its interface. Unlike the `FireiXStreamFormat` and `FireiXTrigger` objects, any change made to the `FireiXFeature` instance is directly passed to the camera, as there is no specific `Save` method.

### **Name property**

#### ***Prototype***

String Name

#### ***Comments***

This is a read-only property that returns the name of the object as a string. This name is the exact same name that can be used to retrieve this `FireiXFeature` object, through the `Feature` property of `FireiXCamera`.

#### ***Visual Basic 6.0 syntax***

```
Dim Name As String  
Name = Feature.Name
```

#### ***C++ syntax***

```
BSTR Name;  
HRESULT hr = pIFeature->get_Name(&Name);
```

### **IsSupported property**

#### ***Prototype***

Boolean IsSupported

#### ***Comments***

This is a read-only property that will return whether this `FireiXFeature` represents a feature supported by the camera.

If this property is False, no other property or method of `FireiXFeature` should be called, otherwise an error would occur.

#### *Visual Basic 6.0 syntax*

```
Dim Supported As Boolean
Supported = Feature.IsSupported
```

#### *C++ syntax*

```
VARIANT_BOOL bSupported;
HRESULT hr = pIFeature->get_IsSupported(&bSupported);
```

## **HasAbsolute property**

#### *Prototype*

```
Boolean HasAbsolute
```

#### *Comments*

This is a read-only property that will return whether this `FireiXFeature` supports setting its value through “Absolute” values.

If this property is False, the `Absolute` property of `FireiXFeature` should not be accessed, otherwise an error would occur.

#### *Visual Basic 6.0 syntax*

```
Dim HasAbsolute As Boolean
HasAbsolute = Feature.HasAbsolute
```

#### *C++ syntax*

```
VARIANT_BOOL bHasAbsolute;
HRESULT hr = pIFeature->get_HasAbsolute(&bHasAbsolute);
```

## **HasOnePush property**

#### *Prototype*

```
Boolean HasOnePush
```

#### *Comments*

This is a read-only property that will return whether this `FireiXFeature` supports setting its value through a “One Push” operation (as defined by the IIDC specification).

If this property is False, the `OnePush` method of `FireiXFeature` should not be called, otherwise an error would occur.

#### *Visual Basic 6.0 syntax*

```
Dim HasOnePush As Boolean
HasOnePush = Feature.HasOnePush
```

#### *C++ syntax*

```
VARIANT_BOOL bHasOnePush;
HRESULT hr = pIFeature->get_HasOnePush(&bHasOnePush);
```

## CanRead property

### Prototype

```
Boolean CanRead
```

### Comments

This is a read-only property that will return whether this `FireiXFeature` supports reading its value.

If this property is `False`, the `Value` property of `FireiXFeature` should not be read, otherwise an error would occur. This does not mean it cannot be set however; the `HasManual` property can be used to determine that.

### Visual Basic 6.0 syntax

```
Dim CanRead As Boolean  
CanRead = Feature.CanRead
```

### C++ syntax

```
VARIANT_BOOL bCanRead;  
HRESULT hr = pIFeature->get_CanRead(&bCanRead);
```

## HasOnOff property

### Prototype

```
Boolean HasOnOff
```

### Comments

This is a read-only property that will return whether this `FireiXFeature` supports turning it on and off entirely.

If this property is `False`, the `Enable` property of `FireiXFeature` should not be accessed, otherwise an error would occur.

### Visual Basic 6.0 syntax

```
Dim HasOnOff As Boolean  
HasOnOff = Feature.HasOnOff
```

### C++ syntax

```
VARIANT_BOOL bHasOnOff;  
HRESULT hr = pIFeature->get_HasOnOff(&bHasOnOff);
```

## HasAuto property

### Prototype

```
Boolean HasAuto
```

### Comments

This is a read-only property that will return whether this `FireiXFeature` supports setting its value automatically, according to the camera shooting circumstances.

If this property is `False`, the `AutoMode` property of `FireiXFeature` should not be accessed, otherwise an error would occur.

### *Visual Basic 6.0 syntax*

```
Dim HasAuto As Boolean
HasAuto = Feature.HasAuto
```

### *C++ syntax*

```
VARIANT_BOOL bHasAuto;
HRESULT hr = pIFeature->get_HasAuto(&bAuto);
```

## **HasManual property**

### *Prototype*

```
Boolean HasManual
```

### *Comments*

This is a read-only property that will return whether this FireiXFeature supports setting its value manually, through the Value property.

If this property is False, the Value property of FireiXFeature should not be set, otherwise an error would occur. It could still be read though; the CanRead property can be used to determine that.

### *Visual Basic 6.0 syntax*

```
Dim HasManual As Boolean
HasManual = Feature.HasManual
```

### *C++ syntax*

```
VARIANT_BOOL bHasManual;
HRESULT hr = pIFeature->get_HasManual(&bHasManual);
```

## **MinValue property**

### *Prototype*

```
Long MinValue
```

### *Comments*

This is a read-only property that will return the minimum value that can be set to this FireiXFeature, through the Value property.

### *Visual Basic 6.0 syntax*

```
Dim MinValue As Long
MinValue = Feature.MinValue
```

### *C++ syntax*

```
LONG lMinValue;
HRESULT hr = pIFeature->get_MinValue(&lMinValue);
```

## **MaxValue property**

### *Prototype*

```
Long MaxValue
```

### Comments

This is a read-only property that will return the maximum value that can be set to this FireiXFeature, through the Value property.

### Visual Basic 6.0 syntax

```
Dim MaxValue As Long
MaxValue = Feature.MaxValue
```

### C++ syntax

```
LONG lMaxValue;
HRESULT hr = pIFeature->get_MaxValue(&lMaxValue);
```

## Absolute property

### Prototype

```
Boolean Absolute
```

### Comments

This is a read/write property that will retrieve or set whether this FireiXFeature will use “absolute” values when setting its value. If set to True, the value of the feature is then read and set through the AbsoluteValue property, instead of the Value property.

This property is not available if HasAbsolute is False.

### Visual Basic 6.0 syntax

```
Dim Absolute As Boolean
Absolute = Feature.Absolute           'get
Feature.Absolute = True              'set
```

### C++ syntax

```
VARIANT_BOOL bAbsolute;
HRESULT hr = pIFeature->get_Absolute(&bAbsolute); //get
hr = pIFeature->put_Absolute(VARIANT_TRUE);      //set
```

## Enabled property

### Prototype

```
Boolean Enabled
```

### Comments

This is a read/write property that will retrieve or set whether this FireiXFeature is on or off. If set to False a call to any of its properties or methods could result in an error.

This property is not available if HasOnOff is False.

### Visual Basic 6.0 syntax

```
Dim Enabled As Boolean
Enabled = Feature.Enabled           'get
Feature.Enabled = True              'set
```

### *C++ syntax*

```
VARIANT_BOOL bEnabled;  
HRESULT hr = pIFeature->get_Enabled(&bEnabled);           //get  
hr = pIFeature->put_Enabled(VARIANT_TRUE);               //set
```

## **AutoMode property**

### *Prototype*

Boolean AutoMode

### *Comments*

This is a read/write property that will retrieve or set whether this FireiXFeature will be setting its value automatically or not. If set to True, the Value property cannot be set, or an error would occur.

This property is not available if HasAuto is False.

### *Visual Basic 6.0 syntax*

```
Dim Auto As Boolean  
Auto = Feature.AutoMode                               'get  
Feature.AutoMode = True                              'set
```

### *C++ syntax*

```
VARIANT_BOOL bAuto;  
HRESULT hr = pIFeature->get_AutoMode(&bAuto);           //get  
hr = pIFeature->put_AutoMode(VARIANT_TRUE);           //set
```

## **Value property**

### *Prototype*

Long Value

### *Comments*

This is a read/write property that will retrieve or set the value of this FireiXFeature. In order to read the value, the CanRead property must be True. In order to set the value, the HasManual property must be True, and the AutoMode property must be False. The value being set must also reside inside the boundaries set by the MinValue and MaxValue properties.

### *Visual Basic 6.0 syntax*

```
Dim Value As Long  
Value = Feature.Value                                 'get  
Feature.Value = Feature.MaxValue                     'set
```

### *C++ syntax*

```
LONG lValue, lMaxValue;  
HRESULT hr = pIFeature->get_Value(&lValue);             //get  
hr = pIFeature = pIFeature->get_MaxValue(&lMaxValue);  
hr = pIFeature->put_Value(lMaxValue);                 //set
```

## HasSoftAbsolute property

### Prototype

```
Boolean HasSoftAbsolute
```

### Comments

This is a read-only property that will return whether this FireiXFeature supports setting its value through “absolute” values, calculated through the API<sup>3</sup>.

If this property is False, the MinAbsoluteValue, MaxAbsoluteValue and SoftAbsolute properties are inaccessible.

### Visual Basic 6.0 syntax

```
Dim HasSoftAbsolute As Boolean  
HasSoftAbsolute = Feature.HasSoftAbsolute
```

### C++ syntax

```
VARIANT_BOOL bHasSoftAbsolute;  
HRESULT hr = pIFeature->get_HasSoftAbsolute(&bHasSoftAbsolute);
```

## SoftAbsolute property

### Prototype

```
Boolean SoftAbsolute
```

### Comments

This is a read/write property that will turn the SoftAbsolute feature on and off, if supported. Trying to set this feature when HasSoftAbsolute is False will result in an error.

If this property is True, the AbsoluteValue property is used to set the feature value instead of the Value property.

### Visual Basic 6.0 syntax

```
Dim SoftAbsolute As Boolean  
SoftAbsolute = Feature.SoftAbsolute  
Feature.SoftAbsolute = True
```

`'get`  
`'set`

### C++ syntax

```
VARIANT_BOOL bSoftAbsolute;  
HRESULT hr = pIFeature->get_SoftAbsolute(&bSoftAbsolute);  
hr = pIFeature->put_AutoMode(VARIANT_TRUE);
```

`//get`  
`//set`

## ValueString property

### Prototype

```
String ValueString
```

---

<sup>3</sup> This feature is called by the various Unibrain APIs “SoftAbsolute”. It is only available on specific cameras, mostly Unibrain models. The camera in effect works in its regular manual values mode; the interface with the program operates in absolute mode, and the underlying API takes care of the value adaptation.

### *Comments*

This is a read/write property that will retrieve or set the value of the feature through a textual representation of it. The string will include the unit of the feature if in absolute mode.

Through this property the value can be read or set either in Absolute, SoftAbsolute or Relative mode, albeit with different formats.

The same limitations as in the Value and AbsoluteValue properties apply.

### *Visual Basic 6.0 syntax*

```
Dim ValueString As String
ValueString = Feature.ValueString           'get
Feature.ValueString = ValueString         'set
```

### *C++ syntax*

```
BSTR ValueString;
HRESULT hr = pIFeature->get_ValueString(&ValueString); //get
hr = pIFeature->put_ValueString(ValueString); //set
```

## **MinValueString property**

### *Prototype*

```
String MinValueString
```

### *Comments*

This is a read-only property that will retrieve the minimum allowed value of the feature as a textual representation. The string will include the unit of the feature if in absolute mode.

The same limitations as in the MinValue and MinAbsoluteValue properties apply.

### *Visual Basic 6.0 syntax*

```
Dim MinValueString As String
MinValueString = Feature.MinValueString
```

### *C++ syntax*

```
BSTR MinValueString;
HRESULT hr = pIFeature->get_MinValueString(&MinValueString);
```

## **MaxValueString property**

### *Prototype*

```
String MaxValueString
```

### *Comments*

This is a read-only property that will retrieve the maximum allowed value of the feature as a textual representation. The string will include the unit of the feature if in absolute mode.

The same limitations as in the MaxValue and MaxAbsoluteValue properties apply.

### *Visual Basic 6.0 syntax*

```
Dim MaxValueString As String
```

```
MaxValueString = Feature.MaxValueString
```

### *++ syntax*

```
BSTR MaxValueString;  
HRESULT hr = pIFeature->get_MaxValueString(&MaxValueString);
```

## **Unit property**

### *Prototype*

```
FireiXFeatureUnit Unit
```

### *Comments*

This is a read-only property that will retrieve the unit of the value of the feature as `FireiXFeatureUnit` enumerated values.

Possible values returned are:

```
fuNone,  
fuFractionPercent,  
fuExposureValue,  
fuKelvin,  
fuDegree,  
fuTime,  
fuDecibel,  
fuFStops,  
fuDistance,  
fuActualPercent
```

### *Visual Basic 6.0 syntax*

```
Dim Unit As FireiXFeatureUnit  
Unit = Feature.Unit
```

### *++ syntax*

```
FireiXFeatureUnit Unit;  
HRESULT hr = pIFeature->get_Unit(&Unit);
```

## **AbsoluteValue property**

### *Prototype*

```
Single AbsoluteValue
```

### *Comments*

This is a read/write property that will retrieve or set the absolute value of this `FireiXFeature`. For this property to be enabled instead of the `Value` property, the `Absolute` or `SoftAbsolute` properties must be `True`. Additionally, the value must reside in the boundaries set by the `MinAbsoluteValue` and `MaxAbsoluteValue` properties.

Since absolute values have fractional parts, this value is represented through a single-precision floating point number.

### *Visual Basic 6.0 syntax*

```
Dim Value As Single
Value = Feature.AbsoluteValue           'get
Feature.Value = Feature.AbsoluteValue  'set
```

### *C++ syntax*

```
float fValue;
HRESULT hr = pIFeature->get_AbsoluteValue(&fValue); //get
hr = pIFeature->put_AbsoluteValue(fValue);         //set
```

## **MinAbsoluteValue property**

### *Prototype*

```
Single MinAbsoluteValue
```

### *Comments*

This is a read-only property that will return the minimum value that can be set to this FireiXFeature, through the AbsoluteValue property.

Since absolute values have fractional parts, this value is represented through a single-precision floating point number.

### *Visual Basic 6.0 syntax*

```
Dim MinValue As Single
MinValue = Feature.MinAbsoluteValue
```

### *C++ syntax*

```
float fMinValue;
HRESULT hr = pIFeature->get_MinAbsoluteValue(&fMinValue);
```

## **MaxAbsoluteValue property**

### *Prototype*

```
Single MaxAbsoluteValue
```

### *Comments*

This is a read-only property that will return the maximum value that can be set to this FireiXFeature, through the AbsoluteValue property.

Since absolute values have fractional parts, this value is represented through a single-precision floating point number.

### *Visual Basic 6.0 syntax*

```
Dim MaxValue As Single
MaxValue = Feature.MaxAbsoluteValue
```

### *C++ syntax*

```
float fMaxValue;
HRESULT hr = pIFeature->get_MaxAbsoluteValue(&fMaxValue);
```

## Reload method

### *Prototype*

```
Reload()
```

### *Comments*

This method can be used to read the current value and absolute value (if applicable, along with the minimum and maximum absolute values) of the camera feature.

It is particularly useful after a OnePush call, or if the camera is in AutoMode, as reading the value through the Value property will not return the current value on the camera, rather the value stored in the FireiXFeature object. This is done for bus bandwidth conservation.

### *Visual Basic 6.0 syntax*

```
Feature.Reload
```

### *C++ syntax*

```
HRESULT hr = pIFeature->Reload();
```

## OnePush method

### *Prototype*

```
OnePush()
```

### *Comments*

This method, when called, will perform a complete one-push operation on the camera for the feature<sup>4</sup>.

This method is available only if HasOnePush is True; calling it otherwise will result in an error.

### *Visual Basic 6.0 syntax*

```
Feature.OnePush
```

### *C++ syntax*

```
HRESULT hr = pIFeature->OnePush();
```

---

<sup>4</sup> A one push operation is two-step: first the register for the one push is set, and then the acknowledge register is repeatedly read until the operation is finished. The OnePush method does these two steps in one call and returns when the one push operation is finished.

## ***EnumFireiXFeatures***

The EnumFireiXFeatures enumerator can be used to iterate between a set of FireiXFeature objects. It cannot be constructed directly, it is obtained through the GetFeaturesEnumerator method of FireiXCamera.

As is the case with EnumFireiXStreamFormats, this object implements only two methods that are sufficient for iterating.

### **Reset method**

#### ***Prototype***

```
Reset()
```

#### ***Comments***

This method will start the iteration from the beginning, i.e., the next call to Next will return the first FireiXFeature.

It is not necessary to call this method on a newly constructed EnumFireiXFeatures object, it will start from the beginning by default.

#### ***Visual Basic 6.0 syntax***

```
Features.Reset
```

#### ***C++ syntax***

```
HRESULT hr = pIFeatures->Reset();
```

### **Next method**

#### ***Prototype***

```
Boolean Next(OUT FireiXFeature Feature)
```

#### ***Comments***

This can be used to obtain the next FireiXFeature in EnumFireiXFeatures. It will return True if another item is available, and False if this was the last item. To start again from the first item, a call to Reset is sufficient.

#### ***Visual Basic 6.0 syntax***

```
Do While Features.Next(Feature) = True  
    ...  
Loop
```

#### ***C++ syntax***

```
VARIANT_BOOL bHasNext;  
HRESULT hr = pIFeatures->Next(&IFeature, &bHasNext);  
while (bHasNext)  
{  
    ...  
    hr = pIFeatures->Next(&IFeature, &bHasNext);  
}
```

## ***FireiXFrame***

The FireiXFrame object contains a single camera frame, allowing the programmer to read and write the image data pixel by pixel. It also provides some useful methods that help manipulate the image, e.g., draw lines, rectangles or print text on the image.

This object cannot be constructed directly; it is constructed and maintained internally by the SDK, and passed through the FrameReceived event of FireiXCamera. It can be considered valid throughout the context of the event handler – but it cannot be stored in memory for later use.

### **GetPixel method**

#### ***Prototype***

```
Long GetPixel(IN Integer X, IN Integer Y)
```

#### ***Comments***

This method returns the color value as a Long, (or OLE\_COLOR) of a given by coordinates pixel of the image buffer.

X and Y must be within the boundaries of the image, starting from 0. The maximum width can be supplied either by the camera (GetCurrentResolution method), or the FireiXStreamFormat properties Width and Height.

#### ***Visual Basic 6.0 syntax***

```
Dim Color As Long  
Color = Frame.GetPixel(0, 0)
```

#### ***C++ syntax***

```
LONG lColor;  
HRESULT hr = pIFrame->GetPixel(0, 0, &lColor);
```

### **SetPixel method**

#### ***Prototype***

```
SetPixel(IN Integer X, IN Integer Y, IN Long Color)
```

#### ***Comments***

This method sets the color value from a Long, (or OLE\_COLOR) to a given by coordinates pixel of the image buffer.

X and Y must be within the boundaries of the image, starting from 0. The maximum width can be supplied either by the camera (GetCurrentResolution method), or the FireiXStreamFormat properties Width and Height.

#### ***Visual Basic 6.0 syntax***

```
Frame.SetPixel 0, 0, 0
```

#### ***C++ syntax***

```
HRESULT hr = pIFrame->SetPixel(0, 0, 0);
```

## GetRGB method

### Prototype

```
GetRGB(  
    IN Integer X,  
    IN Integer Y,  
    OUT Byte Red,  
    OUT Byte Green,  
    OUT Byte Blue  
)
```

### Comments

This method returns the color value as a Red, Green and Blue components of a given by coordinates pixel of the image buffer.

X and Y must be within the boundaries of the image, starting from 0. The maximum width can be supplied either by the camera (GetCurrentResolution method), or the FireiXStreamFormat properties Width and Height.

### Visual Basic 6.0 syntax

```
Dim Red, Green, Blue As Byte  
Frame.GetRGB 0, 0, Red, Green, Blue
```

### C++ syntax

```
BYTE Red, Green, Blue;  
HRESULT hr = pIFrame->GetRGB(0, 0, &Red, &Green, &Blue);
```

## SetRGB method

### Prototype

```
SetRGB(  
    IN Integer X,  
    IN Integer Y,  
    IN Byte Red,  
    IN Byte Green,  
    IN Byte Blue  
)
```

### Comments

This method sets the color value of a given by coordinates pixel of the image buffer, using Red, Green and Blue components.

X and Y must be within the boundaries of the image, starting from 0. The maximum width can be supplied either by the camera (GetCurrentResolution method), or the FireiXStreamFormat properties Width and Height.

### Visual Basic 6.0 syntax

```
Frame.SetRGB 0, 0, 0, 0, 0
```

### C++ syntax

```
HRESULT hr = pIFrame->SetRGB(0, 0, 0, 0, 0);
```

## SaveToFile method

### Prototype

```
Long SaveToFile(IN String Filename)
```

### Comments

This method will save the contents of the Fire1XFrame to disk, as a regular Windows uncompressed BMP file. The Filename must be supplied, with or without the .bmp extension (the SDK will add it if it was omitted).

The return value of SaveToFile is the Win32 error code from the internal save operation. It is 0 if the save was successful.

### Visual Basic 6.0 syntax

```
Dim Win32Err As Long  
Win32Err = Frame.SaveToFile "c:\Frame1.bmp"
```

### C++ syntax

```
LONG lWin32Err;  
HRESULT hr = pIFrame->SaveToFile(_T("c:\Frame1.bmp"), &lWin32Err);
```

## FlipHorizontally method

### Prototype

```
FlipHorizontally()
```

### Comments

This method will invert the frame in-place along the y-axis (horizontally).

### Visual Basic 6.0 syntax

```
Frame.FlipHorizontally
```

### C++ syntax

```
HRESULT hr = pIFrame->FlipHorizontally();
```

## FlipVertically method

### Prototype

```
FlipVertically()
```

### Comments

This method will invert the frame in-place along the x-axis (vertically).

### Visual Basic 6.0 syntax

```
Frame.FlipVertically
```

### C++ syntax

```
HRESULT hr = pIFrame->FlipVertically();
```

## Negative method

### Prototype

```
Negative()
```

### Comments

This method will invert the color information of the frame. The result will be similar to a photographic negative.

### Visual Basic 6.0 syntax

```
Frame.Negative
```

### C++ syntax

```
HRESULT hr = pIFrame->Negative();
```

## ToPicture method

### Prototype

```
IPictureDisp ToPicture()
```

### Comments

This method will return the frame data as an IPictureDisp object. This is a common object type used throughout the COM (and ActiveX) implementation, used for carrying picture data.

### Visual Basic 6.0 syntax

```
Dim Picture As IPictureDisp  
Set Picture = Frame.ToPicture
```

### C++ syntax

```
IPictureDisp* pIPicture;  
HRESULT hr = pIFrame->ToPicture(&pIPicture);
```

## DrawLine method

### Prototype

```
DrawLine(  
    IN Integer FromX,  
    IN Integer FromY,  
    IN Integer ToX,  
    IN Integer ToY,  
    IN Long Color  
)
```

### Comments

This method will draw a line from a point defined by a set of coordinates (FromX, FromY) to another point defined by a set of coordinates (ToX, ToY), using the color defined by a Long number (OLE\_COLOR).

The coordinates given as the 4 parameters must reside in the frame boundaries, otherwise an error will occur.

### *Visual Basic 6.0 syntax*

```
Frame.DrawLine 0, 0, 50, 50, 0
```

### *C++ syntax*

```
HRESULT hr = pIFrame->DrawLine(0, 0, 50, 50, 0);
```

## **DrawString method**

### *Prototype*

```
DrawString(  
    IN String Text,  
    IN Integer X,  
    IN Integer Y,  
    IN IFontDisp Font,  
    IN Long Color  
)
```

### *Comments*

This method will draw a string of text on a point on the frame defined by a set of coordinates (X, Y), using the color defined by a Long number (OLE\_COLOR), and a valid IFontDisp object.

The coordinates given as the 2 parameters must reside in the frame boundaries, otherwise an error will occur.

### *Visual Basic 6.0 syntax*

```
Frame.DrawString "Hello World", 0, 0, Me.Font, 0
```

### *C++ syntax*

```
HRESULT hr = pIFrame->DrawString(  
    _T("Hello World"),  
    0, 0,  
    pIFont,  
    0);
```

## **DrawRectangle method**

### *Prototype*

```
DrawRectangle(  
    IN Integer X,  
    IN Integer Y,  
    IN Integer Width,  
    IN Integer Height,  
    IN Long Color,  
    IN Boolean Filled  
)
```

### *Comments*

This method will draw a rectangle, with its top-left corner being at a point defined by a set of coordinates (X, Y) having a specific Width and Height and using the color defined by a Long number (OLE\_COLOR). Additionally, it can be specified whether this rectangle will be empty (just its outline drawn) or filled.

The coordinates defined by the 4 parameters given must reside in the frame boundaries, otherwise an error will occur.

#### *Visual Basic 6.0 syntax*

```
Frame.DrawRectangle 0, 0, 50, 50, 0, False
```

#### *C++ syntax*

```
HRESULT hr = pIFrame->DrawRectangle(0, 0, 50, 50, 0, VARIANT_FALSE);
```

## **DrawLineRGB method**

#### *Prototype*

```
DrawLineRGB(  
    IN Integer FromX,  
    IN Integer FromY,  
    IN Integer ToX,  
    IN Integer ToY,  
    IN Byte Red,  
    IN Byte Green,  
    IN Byte Blue  
)
```

#### *Comments*

This method will draw a line from a point defined by a set of coordinates (FromX, FromY) to another point defined by a set of coordinates (ToX, ToY), using the color defined by its Red, Green and Blue components.

The coordinates given as the 4 parameters must reside in the frame boundaries, otherwise an error will occur.

#### *Visual Basic 6.0 syntax*

```
Frame.DrawLineRGB 0, 0, 50, 50, 0, 0, 0
```

#### *C++ syntax*

```
HRESULT hr = pIFrame->DrawLineRGB(0, 0, 50, 50, 0, 0, 0);
```

## **DrawStringRGB method**

#### *Prototype*

```
DrawString(  
    IN String Text,  
    IN Integer X,  
    IN Integer Y,  
    IN IFontDisp Font,  
    IN Byte Red,  
    IN Byte Green,  
    IN Byte Blue  
)
```

#### *Comments*

This method will draw a string of text on a point on the frame defined by a set of coordinates (X, Y), using the color defined by its Red, Green and Blue components, and a valid IFontDisp object.

The coordinates given as the 2 parameters must reside in the frame boundaries, otherwise an error will occur.

#### *Visual Basic 6.0 syntax*

```
Frame.DrawStringRGB "Hello World", 0, 0, Me.Font, 0, 0, 0
```

#### *C++ syntax*

```
HRESULT hr = pIFrame->DrawStringRGB(  
    _T("Hello World"),  
    0, 0,  
    pIFont,  
    0, 0, 0);
```

## **DrawRectangleRGB method**

#### *Prototype*

```
DrawRectangle(  
    IN Integer X,  
    IN Integer Y,  
    IN Integer Width,  
    IN Integer Height,  
    IN Byte Red,  
    IN Byte Green,  
    IN Byte Blue,  
    IN Boolean Filled  
)
```

#### *Comments*

This method will draw a rectangle, with its top-left corner being at a point defined by a set of coordinates (X, Y) having a specific Width and Height and using the color defined by its Red, Green and Blue components. Additionally, it can be specified whether this rectangle will be empty (just its outline drawn) or filled.

The coordinates defined by the 4 parameters given must reside in the frame boundaries, otherwise an error will occur.

#### *Visual Basic 6.0 syntax*

```
Frame.DrawRectangleRGB 0, 0, 50, 50, 0, 0, 0, False
```

#### *C++ syntax*

```
HRESULT hr = pIFrame->DrawRectangleRGB(  
    0, 0,  
    50, 50,  
    0, 0, 0,  
    VARIANT_FALSE);
```

## ***FireiXCamera***

The `FireiXCamera` object encapsulates all functionality pertaining to the camera itself; therefore it could be considered to be the “central” object of the SDK.

It acts as a factory for many other objects: `FireiXFeature`, `FireiXStreamFormat` and `FireiXTrigger`, along with the two enumerators, `EnumFireiXFeatures` and `EnumFireiXStreamFormats`.

It cannot be constructed directly however; the `FireiXManager` object is responsible for constructing a `FireiXCamera` instance.

Besides having a great number of properties and methods, it also implements three events: `FrameReceived`, `BufferReceived` and `DeviceRemoved`. The first, if handled, will be fired every time a new frame is received by the SDK; the second will be fired every time a new frame is received by the SDK, but before converting it to a display-compatible format; and the third will be fired if the camera is physically removed from the system.

### **BufferMode property**

#### ***Prototype***

```
Boolean BufferMode
```

#### ***Comments***

This is a read/write property that can be used to enable/disable buffer-mode operation. This mode affects which of the `FrameReceived` or `BufferReceived` events will be fired when the camera is capturing frames. When set to true, `BufferReceived` will be fired if handled, otherwise `FrameReceived` will be fired if handled.

The default value is false. Changing this property while the camera is running might lead to unexpected behavior.

#### ***Visual Basic 6.0 syntax***

```
Camera.BufferMode = True
```

#### ***C++ syntax***

```
VARIANT_BOOL BufferMode;  
HRESULT hr = pICamera->get_BufferMode(&BufferMode);
```

### **GUID property**

#### ***Prototype***

```
FireiGUID GUID
```

#### ***Comments***

This is a read-only property that can be used to retrieve the GUID of the camera. It constructs and initializes a `FireiGUID` object internally and then returns it.

#### ***Visual Basic 6.0 syntax***

```
Dim GUID As FireiGUID
```

```
Set GUID = Camera.GUID
```

#### *C++ syntax*

```
FireiGUID* pIGUID;  
HRESULT hr = pICamera->get_GUID(&pIGUID);
```

## **Vendor property**

#### *Prototype*

```
String Vendor
```

#### *Comments*

This is a read-only property that can be used to retrieve the vendor name of the camera, as a string.

#### *Visual Basic 6.0 syntax*

```
Dim VendorName As String  
VendorName = Camera.Vendor
```

#### *C++ syntax*

```
BSTR VendorName;  
HRESULT hr = pICamera->get_Vendor(&VendorName);
```

## **Model property**

#### *Prototype*

```
String Model
```

#### *Comments*

This is a read-only property that can be used to retrieve the model name of the camera, as a string.

#### *Visual Basic 6.0 syntax*

```
Dim ModelName As String  
ModelName = Camera.Model
```

#### *C++ syntax*

```
BSTR ModelName;  
HRESULT hr = pICamera->get_Model(&ModelName);
```

## **Serial property**

#### *Prototype*

```
Long Serial
```

#### *Comments*

This is a read-only property that can be used to retrieve the serial number of the camera, as a Long.

#### *Visual Basic 6.0 syntax*

```
Dim SerialNo As Long  
SerialNo = Camera.Serial
```

#### *C++ syntax*

```
LONG lSerialNo;
```

```
HRESULT hr = pICamera->get_Serial(&lSerialNo);
```

## FriendlyName property

### Prototype

```
String FriendlyName
```

### Comments

This is a read-only property that can be used to retrieve an SDK-constructed “friendly name” for the camera, as a Long. This “friendly name” is a combination of the vendor name, the model name and the serial number of the camera.

A typical friendly name is similar to:

```
Unibrain Fire-i 1.2 40045266
```

### Visual Basic 6.0 syntax

```
Dim FriendlyName As String  
FriendlyName = Camera.FriendlyName
```

### C++ syntax

```
BSTR FriendlyName;  
HRESULT hr = pICamera->get_FriendlyName(&FriendlyName);
```

## StreamFormat property

### Prototype

```
FireiXStreamFormat StreamFormat
```

### Comments

This is a read/write property that will retrieve or set the currently selected FireiXStreamFormat of this FireiXCamera. Through this FireiXStreamFormat, all the streaming format functionality of the camera is exposed to the programmer.

The “set” operation of this property is the equivalent of calling the Save method of FireiXStreamFormat.

### Visual Basic 6.0 syntax

```
Dim StreamFormat As FireiXStreamFormat  
Set StreamFormat = Camera.StreamFormat  
Camera.StreamFormat = StreamFormat
```

```
'get  
'set
```

### C++ syntax

```
FireiXStreamFormat* pIStreamFormat;  
HRESULT hr = pICamera->get_StreamFormat(&pIStreamFormat); //get  
hr = pICamera->put_StreamFormat(pIStreamFormat); //set
```

## Trigger property

### Prototype

```
FireiXTrigger Trigger
```

### Comments

This is a read/write property that will retrieve or set the currently selected FireiXTrigger of this FireiXCamera. Through this FireiXTrigger, all the trigger functionality of the camera is exposed to the programmer.

The “set” operation of this property is the equivalent of calling the Save method of FireiXTrigger.

### Visual Basic 6.0 syntax

```
Dim Trigger As FireiXTrigger
Set Trigger = Camera.Trigger           'get
Camera.Trigger = Trigger               'set
```

### C++ syntax

```
FireiXTrigger* pITrigger;
HRESULT hr = pICamera->get_Trigger(&pITrigger); //get
hr = pICamera->put_Trigger(pITrigger);        //set
```

## AutoExposure property

### Prototype

```
FireiXFeature AutoExposure
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the AutoExposure feature of the camera.

### Visual Basic 6.0 syntax

```
Dim AutoExposure As FireiXFeature
Set AutoExposure = Camera.AutoExposure
```

### C++ syntax

```
FireiXFeature* pIAutoExposure;
HRESULT hr = pICamera->get_AutoExposure(&pIAutoExposure);
```

## Shutter property

### Prototype

```
FireiXFeature Shutter
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Shutter feature of the camera.

### Visual Basic 6.0 syntax

```
Dim Shutter As FireiXFeature
Set Shutter = Camera.Shutter
```

### C++ syntax

```
FireiXFeature* pIShutter;
HRESULT hr = pICamera->get_Shutter(&pIShutter);
```

## Gain property

### *Prototype*

```
FireiXFeature Gain
```

### *Comments*

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Gain feature of the camera.

### *Visual Basic 6.0 syntax*

```
Dim Gain As FireiXFeature  
Set Gain = Camera.Gain
```

### *C++ syntax*

```
FireiXFeature* pIGain;  
HRESULT hr = pICamera->get_Gain(&pIGain);
```

## Iris property

### *Prototype*

```
FireiXFeature Iris
```

### *Comments*

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Iris feature of the camera.

### *Visual Basic 6.0 syntax*

```
Dim Iris As FireiXFeature  
Set Iris = Camera.Iris
```

### *C++ syntax*

```
FireiXFeature* pIIris;  
HRESULT hr = pICamera->get_Iris(&pIIris);
```

## ColorUB property

### *Prototype*

```
FireiXFeature ColorUB
```

### *Comments*

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the U/B component of the Color feature of the camera.

### *Visual Basic 6.0 syntax*

```
Dim ColorUB As FireiXFeature  
Set ColorUB = Camera.ColorUB
```

### *C++ syntax*

```
FireiXFeature* pIColorUB;  
HRESULT hr = pICamera->get_ColorUB(&pIColorUB);
```

## ColorVR property

### Prototype

```
FireiXFeature ColorVR
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the V/R component of the Color feature of the camera.

### Visual Basic 6.0 syntax

```
Dim ColorVR As FireiXFeature  
Set ColorVR = Camera.ColorVR
```

### C++ syntax

```
FireiXFeature* pIColorVR;  
HRESULT hr = pICamera->get_ColorVR(&pIColorVR);
```

## Hue property

### Prototype

```
FireiXFeature Hue
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Hue feature of the camera.

### Visual Basic 6.0 syntax

```
Dim Hue As FireiXFeature  
Set Hue = Camera.Hue
```

### C++ syntax

```
FireiXFeature* pIHue;  
HRESULT hr = pICamera->get_Hue(&pIHue);
```

## Saturation property

### Prototype

```
FireiXFeature Saturation
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Saturation feature of the camera.

### Visual Basic 6.0 syntax

```
Dim Saturation As FireiXFeature  
Set Saturation = Camera.Saturation
```

### C++ syntax

```
FireiXFeature* pISaturation;  
HRESULT hr = pICamera->get_Saturation(&pISaturation);
```

## Focus property

### *Prototype*

```
FireiXFeature Focus
```

### *Comments*

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Focus feature of the camera.

### *Visual Basic 6.0 syntax*

```
Dim Focus As FireiXFeature  
Set Focus = Camera.Focus
```

### *C++ syntax*

```
FireiXFeature* pIFocus;  
HRESULT hr = pICamera->get_Focus(&pIFocus);
```

## Zoom property

### *Prototype*

```
FireiXFeature Zoom
```

### *Comments*

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Zoom feature of the camera.

### *Visual Basic 6.0 syntax*

```
Dim Zoom As FireiXFeature  
Set Zoom = Camera.Zoom
```

### *C++ syntax*

```
FireiXFeature* pIZoom;  
HRESULT hr = pICamera->get_Zoom(&pIZoom);
```

## Brightness property

### *Prototype*

```
FireiXFeature Brightness
```

### *Comments*

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Brightness feature of the camera.

### *Visual Basic 6.0 syntax*

```
Dim Brightness As FireiXFeature  
Set Brightness = Camera.Brightness
```

### *C++ syntax*

```
FireiXFeature* pIBrightness;  
HRESULT hr = pICamera->get_Brightness(&pIBrightness);
```

## Sharpness property

### Prototype

```
FireiXFeature Sharpness
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Sharpness feature of the camera.

### Visual Basic 6.0 syntax

```
Dim Sharpness As FireiXFeature  
Set Sharpness = Camera.Sharpness
```

### C++ syntax

```
FireiXFeature* pISharpness;  
HRESULT hr = pICamera->get_Sharpness(&pISharpness);
```

## Gamma property

### Prototype

```
FireiXFeature Gamma
```

### Comments

This is a read-only property that can be used to retrieve directly the FireiXFeature object for the Gamma feature of the camera.

### Visual Basic 6.0 syntax

```
Dim Gamma As FireiXFeature  
Set Gamma = Camera.Gamma
```

### C++ syntax

```
FireiXFeature* pIGamma;  
HRESULT hr = pICamera->get_Gamma(&pIGamma);
```

## Feature property

### Prototype

```
FireiXFeature Feature(IN String FeatureName)
```

### Comments

This is a read-only property that can be used to retrieve the FireiXFeature object of any of features of the camera.

It takes the name (as a string) of the feature to locate the desired FireiXFeature object. This name is the same as the one returned by the Name property of FireiXFeature. If the parameter passed is not a recognized feature name, an error will occur.

### Visual Basic 6.0 syntax

```
Dim Feature As FireiXFeature  
Set Feature = Camera.Feature("Shutter")
```

### *C++ syntax*

```
FireiXFeature* pIFeature;  
HRESULT hr = pICamera->get_Feature(_T("Shutter"), &pIFeature);
```

## **RawConversion property**

### *Prototype*

```
FireiXRawConversion RawConversion
```

### *Comments*

This is a read/write property that can be used to retrieve or set the Bayer conversion method that will be employed in case it is set on the streaming format.

The Fire-iX SDK uses the Bayer conversion algorithms implemented in the Firei.dll and DirectShow APIs as they are, benefitting automatically from future performance updates in them.

Possible values of the `FireiXRawConversion` enumerated value are:

```
rcNearestNeighbor,  
rcBilinearInterpolation,  
rcSmoothHueTransition
```

The three implemented algorithms are ordered “Best Performance → Best Quality”.

`rcBilinearInterpolation` is the default setting.

Please note that the `rcSmoothHueTransition` algorithm is only implemented in the Firei.dll API; if set while using the DirectShow API, an error will occur.

### *Visual Basic 6.0 syntax*

```
Dim RawConversion As FireiXRawConversion  
RawConversion = Camera.RawConversion 'get  
Camera.RawConversion = rcBilinearInterpolation 'set
```

### *C++ syntax*

```
FireiXRawConversion RawConversion;  
HRESULT hr = pICamera->get_RawConversion(&RawConversion); //get  
hr = pICamera->put_RawConversion(rcBilinearInterpolation); //set
```

## **Register property**

### *Prototype*

```
FireiXRegister Register(IN Long Offset)
```

### *Comments*

This is a read/write property that can be used to retrieve or set any register of the camera, given its offset as a parameter.

The get operation will read the value from the camera and initialize a `FireiXFeature` object, returning it to the programmer. The set operation will write the value to the camera, given a valid `FireiXFeature` object. If for some reason the camera rejects the read or write request, an error will occur (the exception information will contain the nature of the error).

Please note that the offset parameter is based at 0 – for accessing the command registers of the camera (normally based at hex F0000000) the `CommandRegister` parameter can be used instead.

#### *Visual Basic 6.0 syntax*

```
Dim Register As FireiXRegister
Register = Camera.Register(&H404)           'get
Camera.Register(&H404) = Register          'set
```

#### *C++ syntax*

```
FireiXRegister* pIRegister;
HRESULT hr = pICamera->get_Register(0x404, &pIRegister); //get
hr = pICamera->put_Register(0x404, pIRegister);         //set
```

## **CommandRegister property**

#### *Prototype*

```
FireiXRegister CommandRegister(IN Long Offset)
```

#### *Comments*

This is a read/write property that can be used to retrieve or set any command register of the camera, given its offset as a parameter. As “command register” is defined as a register with its offset based on the command-base register offset (normally hex F0000000).

The get operation will read the value from the camera and initialize a `FireiXFeature` object, returning it to the programmer. The set operation will write the value to the camera, given a valid `FireiXFeature` object. If for some reason the camera rejects the read or write request, an error will occur (the exception information will contain the nature of the error).

The `CommandRegister` property has the exact same effect as the `Register` property, if the command-base offset is added to the offset parameter. Since this offset is theoretically variable (camera-specific), for command registers it is best to use the `CommandRegister` property.

#### *Visual Basic 6.0 syntax*

```
Dim Register As FireiXRegister
Register = Camera.CommandRegister(&H504)           'get
Camera.CommandRegister(&H504) = Register          'set
```

#### *C++ syntax*

```
FireiXRegister* pIRegister;
HRESULT hr =
    pICamera->get_CommandRegister(0x504, &pIRegister); //get
hr = pICamera->put_CommandRegister(0x504, pIRegister); //set
```

## **Icon property**

#### *Prototype*

```
IPictureDisp Icon
```

### *Comments*

This is a read-only property that can be used to retrieve an IPictureDisp object of icon-type, depicting the camera itself. This picture is derived from an internal database of camera pictures that the Unibrain APIs maintain and is selected automatically, using the camera vendor and model.

### *Visual Basic 6.0 syntax*

```
Dim Icon As IPictureDisp  
Icon = Camera.Icon
```

### *C++ syntax*

```
IPictureDisp* pIIcon;  
HRESULT hr = pICamera->get_Icon(&pIIcon);
```

## **NumOfMemoryPresets property**

### *Prototype*

```
Long NumOfMemoryPresets
```

### *Comments*

This is a read-only property that can be used to retrieve the number of available memory presets of the camera.

If this number is 0, the camera does not support memory presets (rendering the SaveToMemory and LoadFromMemory methods inaccessible).

### *Visual Basic 6.0 syntax*

```
Dim NumOfMemoryPresets As Long  
NumOfMemoryPresets = Camera.NumOfMemoryPresets
```

### *C++ syntax*

```
LONG lNumOfMemoryPresets;  
HRESULT hr = pICamera->get_NumOfMemoryPresets(&lNumOfMemoryPresets);
```

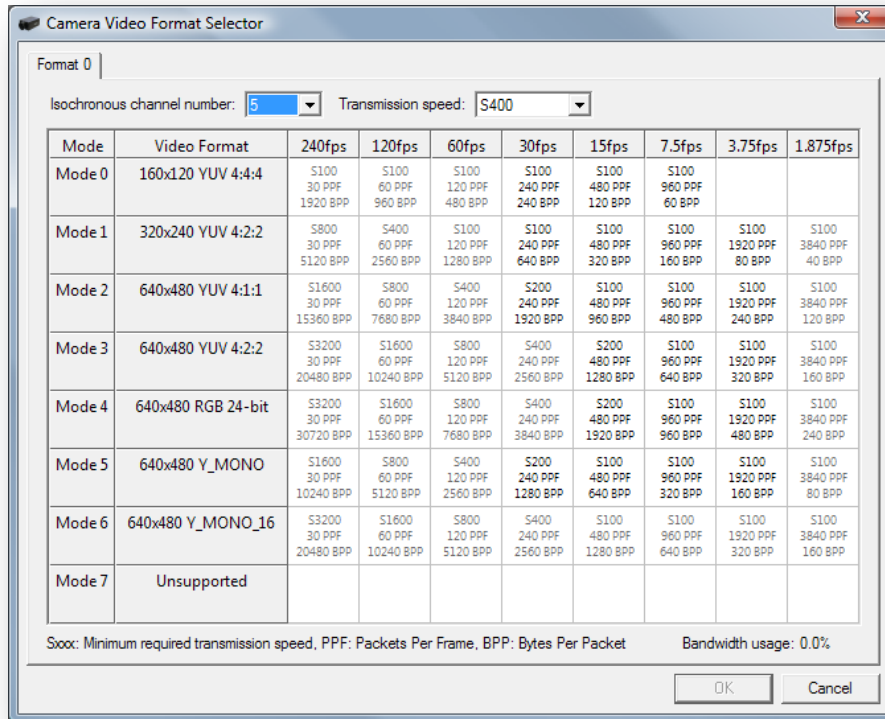
## **SelectStreamFormat method**

### *Prototype*

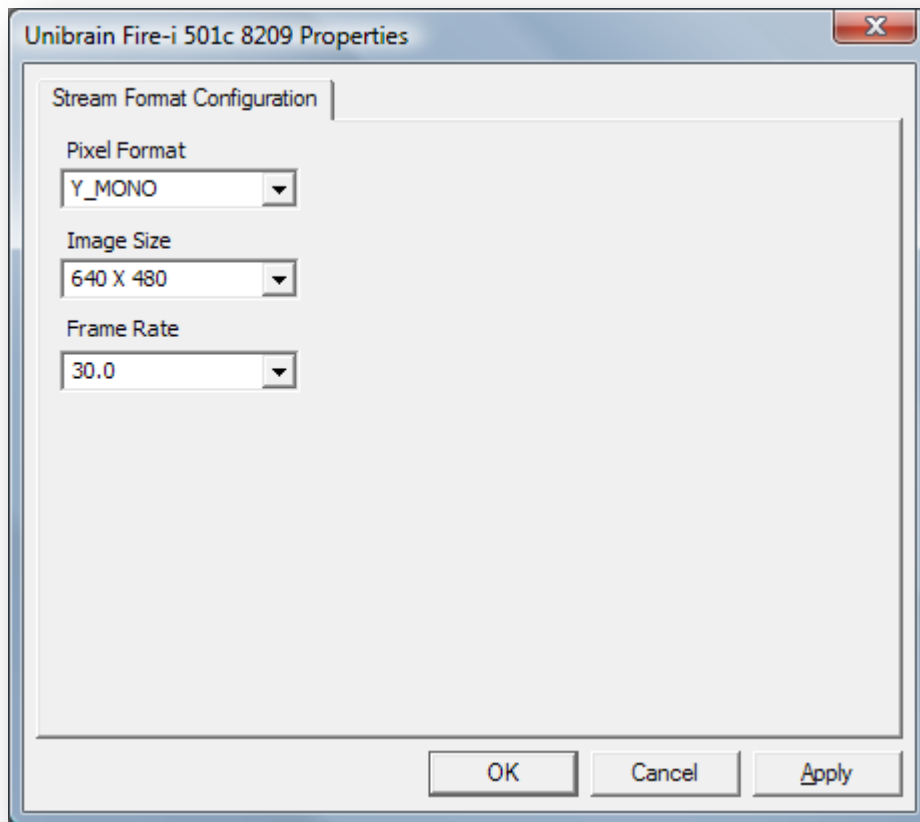
```
SelectStreamFormat()
```

### *Comments*

This method when called will bring up a “Stream Format Selector” dialog, as constructed and maintained internally by the APIs. Through this dialog, user selection of a streaming format is possible. The appearance of this dialog varies depending on the underlying API selected. If Firei.dll is currently running, the dialog will look similar to:



If on the other hand the DirectShow API is currently running, the dialog will look similar to:



Upon the return of the method, the streaming format that the user selected will be also set on the camera automatically. It can be retrieved by the programmer, if desired, through the `StreamFormat` property. If the user did not select a format, pressing “Cancel”, there is no change made to the camera.

#### *Visual Basic 6.0 syntax*

```
Camera.SelectStreamFormat
```

#### *C++ syntax*

```
HRESULT hr = pICamera->SelectStreamFormat();
```

### **AttachPreviewCtrl method**

#### *Prototype*

```
AttachPreviewCtrl(IN FireiXPreviewCtrl PreviewCtrl)
```

#### *Comments*

This method can be used to attach a `FireiXPreviewCtrl` object to this `FireiXCamera`, for video preview purposes.

It has the exact same effect as calling the `AttachCamera` method of `FireiXPreviewCtrl`.

### *Visual Basic 6.0 syntax*

```
Camera.AttachPreviewCtrl FireiXPreviewCtrl1
```

### *C++ syntax*

```
HRESULT hr = pICamera->AttachPreviewCtrl(pIFireiXPreviewCtrl);
```

## **Run method**

### *Prototype*

```
Run()
```

### *Comments*

This method will start the streaming of the camera.

If the FireiXCamera object is attached to a FireiXPreviewCtrl, the preview on that control will also start.

If the camera is already running, a call to Run will have no effect<sup>5</sup>. If the camera for some reason cannot start, or the preview window (if applicable) is invalid in some way, an error will occur.

### *Visual Basic 6.0 syntax*

```
Camera.Run
```

### *C++ syntax*

```
HRESULT hr = pICamera->Run();
```

## **Stop method**

### *Prototype*

```
Stop()
```

### *Comments*

This method will stop the streaming of the camera.

If the FireiXCamera object is attached to a FireiXPreviewCtrl, the preview on that control will also stop.

If the camera is not running, a call to Stop will have no effect. If the camera for some reason cannot stop, an error will occur.

### *Visual Basic 6.0 syntax*

```
Camera.Stop
```

### *C++ syntax*

```
HRESULT hr = pICamera->Stop();
```

---

<sup>5</sup> This is not entirely true; if the camera was started while not being attached to a FireiXPreviewCtrl, then attached and Run was called again, the preview will start.

## IsRunning method

### Prototype

```
Boolean IsRunning()
```

### Comments

This method will return whether the camera is currently running.

Please note that IsRunning will return True even if the camera was not started during runtime (i.e., it was running before the program was run).

### Visual Basic 6.0 syntax

```
Dim Running As Boolean  
Running = Camera.IsRunning
```

### C++ syntax

```
VARIANT_BOOL bRunning;  
HRESULT hr = pICamera->IsRunning(&bRunning);
```

## GetStreamFormatsEnumerator method

### Prototype

```
EnumFireiXStreamFormats GetStreamFormatsEnumerator()
```

### Comments

This method can be used to retrieve the EnumFireiXStreamFormats object, the FireiXStreamFormat enumerator object.

The EnumFireiXStreamFormats enumerator that is returned contains only the supported formats of the specific camera at that specific time. Since the actual speed of the bus the camera is connected to can vary depending on various parameters, the supported formats will also vary, depending on the speed of the bus (besides the make and model of the camera).

For more information on how to iterate between the FireiXStreamFormat objects contained in EnumFireiXStreamFormats, please refer to the EnumFireiXStreamFormats section of this text.

### Visual Basic 6.0 syntax

```
Dim StreamFormats As EnumFireiXStreamFormats  
Set StreamFormats = Camera.GetStreamFormatsEnumerator
```

### C++ syntax

```
IEnumFireiXStreamFormats* pIStreamFormats;  
HRESULT hr = pICamera->GetStreamFormatsEnumerator(&pIStreamFormats);
```

## DisplayProperties method

### Prototype

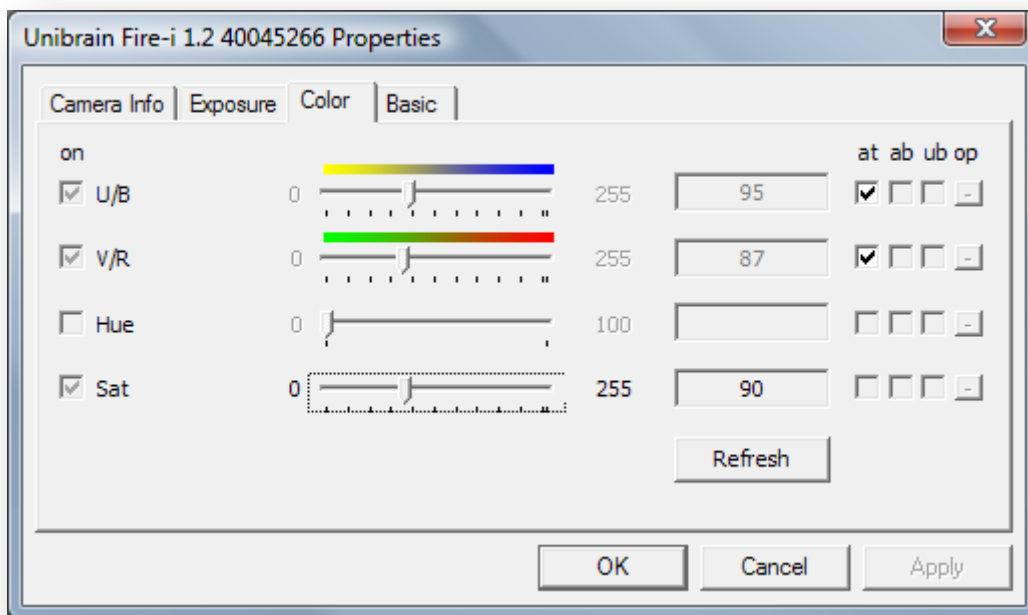
```
DisplayProperties()
```

### Comments

This method when called will bring up a “Display Properties” dialog for the camera. This dialog can then be used to set the various feature values of the camera.

Unlike the other UI-driven methods of the SDK (namely `SelectCamera` and `SelectStreamFormat`), this dialog has an immediate effect on the camera as the values and settings of the features are changed. Additionally, the method opens up the dialog and returns immediately. The dialog will remain open until the user closes it, affecting the camera whether it is running or not.

The presented dialog is almost exactly the same in appearance, regardless if `DirectShow` or `Firei.dll` is the current selected underlying API, and it looks similar to:



### Visual Basic 6.0 syntax

```
Camera.DisplayProperties
```

### C++ syntax

```
HRESULT hr = pICamera->DisplayProperties();
```

## IsFeatureSupported method

### Prototype

```
Boolean IsFeatureSupported(IN String FeatureName)
```

### Comments

This method will return whether a given feature is supported. The feature parameter is defined as a String, containing the name of the feature. This name is the same as the one the `Name` property of `FireiXFeature` would return.

### *Visual Basic 6.0 syntax*

```
Dim Supported As Boolean
Supported = Camera.IsFeatureSupported("Shutter")
```

### *C++ syntax*

```
VARIANT_BOOL bSupported;
HRESULT hr =
    pICamera->IsFeatureSupported(_T("Shutter"), &bSupported);
```

## **GetFeaturesEnumerator method**

### *Prototype*

```
EnumFireiXFeatures GetFeaturesEnumerator(
    IN Boolean SupportedOnly,
    IN FireiXFeatureGroup FeatureGroup
)
```

### *Comments*

This method will construct and return an EnumFireiXFeatures object. This enumerator will contain the FireiXFeature objects, depending on the parameters pass to GetFeaturesEnumerator. The first parameter will toggle whether all features will be returned, or only the ones supported by the camera. Additionally, since the features are divided in 3 groups (Exposure, Color, Basic), through the FeatureGroup parameter, further reduction on the returned set can be achieved.

The possible values for the FeatureGroup parameter are:

```
fgAll,
fgExposure,
fgColor,
fgBasic
```

If fgAll is selected, all features will participate in the enumerator, otherwise only the ones defined by the FireiXFeatureGroup value.

For more information on how to iterate between the FireiXFeature objects contained in EnumFireiXFeatures, please refer to the EnumFireiXFeatures section of this text.

### *Visual Basic 6.0 syntax*

```
Dim Features As EnumFireiXFeatures
Set Features = Camera.GetFeaturesEnumerator(True, fgAll)
```

### *C++ syntax*

```
IEnumFireiXFeatures* pIFeatures;
HRESULT hr = pICamera->GetFeaturesEnumerator(
    VARIANT_TRUE,
    fgAll,
    &pIFeatures
);
```

## GetCurrentResolution method

### Prototype

```
GetCurrentResolution(OUT Integer Width, OUT Integer Height)
```

### Comments

This method will return (by reference) the width and height of the camera stream format resolution in pixels.

### Visual Basic 6.0 syntax

```
Dim Width, Height As Integer  
Camera.GetCurrentResolution Width, Height
```

### C++ syntax

```
SHORT shWidth, shHeight;  
HRESULT hr = pICamera->GetCurrentResolution(&shWidth, &shHeight);
```

## GetCameraPhoto method

### Prototype

```
IPictureDisp GetCameraPhoto(  
    IN Integer EdgeSize,  
    IN Byte bgRed,  
    IN Byte bgGreen,  
    IN Byte bgBlue,  
    IN Byte bgAlpha  
)
```

### Comments

This method constructs and returns an IPictureDisp object, of type bitmap, that contains a photograph of the camera connected. This photograph is derived from an internal database that Unibrain maintains, and the SDK utilizes.

The resulting picture will be square (width equal height); the size of each edge can be defined. Additionally, the programmer can choose the background color of the picture (for rendering transparently on a control), in all 4 color components, Red, Green, Blue and Alpha.

### Visual Basic 6.0 syntax

```
Dim Photo As IPictureDisp;  
Photo = Camera.GetCameraPhoto(80, 255, 255, 255, 255)
```

### C++ syntax

```
IPictureDisp* pIPhoto;  
HRESULT hr = pICamera->GetCameraPhoto(80, 255, 255, 255, 255, &pIPhoto);
```

## SaveToMemory method

### Prototype

```
SaveToMemory(IN Byte ChannelNumber)
```

### *Comments*

This method will save all current features to a memory preset channel on the camera. Since channel 0 is reserved for the camera default values, it cannot be used as a parameter in SaveToMemory.

In order to ascertain what the maximum channel number allowed is, the NumOfMemoryPresets property can be used: in effect, the number representing the NumOfMemoryPresets value is the maximum allowed value for SaveToMemory.

If for any reason the camera cannot save the settings, an error will occur.

### *Visual Basic 6.0 syntax*

```
Camera.SaveToMemory 1
```

### *C++ syntax*

```
HRESULT hr = pICamera->SaveToMemory(1);
```

## **LoadFromMemory method**

### *Prototype*

```
LoadFromMemory(IN Byte ChannelNumber)
```

### *Comments*

This method will load all current features from a memory preset channel of the camera. Since channel 0 is reserved for the camera default values, it cannot be used as a parameter in LoadFromMemory.

In order to ascertain what the maximum channel number allowed is, the NumOfMemoryPresets property can be used: in effect, the number representing the NumOfMemoryPresets value is the maximum allowed value for LoadFromMemory.

If for any reason the camera cannot load the settings, an error will occur.

### *Visual Basic 6.0 syntax*

```
Camera.LoadFromMemory 1
```

### *C++ syntax*

```
HRESULT hr = pICamera->LoadFromMemory(1);
```

## **SaveToXML method**

### *Prototype*

```
SaveToXML(IN String Filename)
```

### *Comments*

This method will save all current features to a file on disk, in XML format. The parameter passed represents the filename to save as, and it cannot be empty.

The resulting XML file has 100% compatible format with the XML files saved and loaded from the other Unibrain tools, like FireIIDC and Fire-i Application.

If for any reason the file cannot be written to disk, an error will occur.

### *Visual Basic 6.0 syntax*

```
Camera.SaveToXML "c:\settings.xml"
```

### *C++ syntax*

```
HRESULT hr = pICamera->SaveToXML(_T("c:\settings.xml"));
```

## **LoadFromXML method**

### *Prototype*

```
LoadFromXML(IN String Filename)
```

### *Comments*

This method will load all features from an XML file on disk, and set them to the camera. The parameter passed represents the filename to load from, and it cannot be empty.

The XML file being loaded can be the one saved with SaveToXML, or any other XML files created by the other Unibrain tools, like FireIDC and Fire-i Application, since these are 100% compatible.

If for any reason the settings cannot be loaded or set to the camera, an error will occur.

### *Visual Basic 6.0 syntax*

```
Camera.LoadFromXML "c:\settings.xml"
```

### *C++ syntax*

```
HRESULT hr = pICamera->LoadFromXML(_T("c:\settings.xml"));
```

## **RetrieveStreamFormat method**

### *Prototype*

```
FireiXStreamFormat RetrieveStreamFormat(  
    IN FireiXPixelFormat PixelFormat,  
    IN FireiXResolution Resolution  
)
```

### *Comments*

This method will return a FireiXStreamFormat object that has the given FireiXPixelFormat and FireiXResolution attributes, provided it is supported by the camera.

There is only a single FireiXStreamFormat object having the same FireiXPixelFormat and FireiXResolution values.

### *Visual Basic 6.0 syntax*

```
Dim StreamFormat As FireiXStreamFormat  
Set StreamFormat = Camera.RetrieveStreamFormat(pfY_MONO, res320x240)
```

### *C++ syntax*

```
IFireiXStreamFormat* pIStreamFormat;  
HRESULT hr = pICamera->RetrieveStreamFormat(  
    pfY_MONO, res320x240, &pIStreamFormat);
```

## RetrieveStreamFormatFromIdentifier method

### Prototype

```
FireiXStreamFormat RetrieveStreamFormatFromIdentifier(  
    IN Long Identifier  
)
```

### Comments

This method will return a FireiXStreamFormat object that has the given unique identifier. This identifier can be retrieved by the Identifier property of FireiXStreamFormat only.

If the identifier given is not valid, or if it points to a FireiXStreamFormat that is not supported by the camera, an error will occur.

### Visual Basic 6.0 syntax

```
Dim Identifier As Long 'Identifier value retrieved by any means  
Dim StreamFormat As FireiXStreamFormat  
Set StreamFormat = _  
    Camera.RetrieveStreamFormatFromIdentifier(Identifier)
```

### C++ syntax

```
IFireiXStreamFormat* pIStreamFormat;  
LONG lIdentifier; // lIdentifier value retrieved by any means  
HRESULT hr = pICamera->RetrieveStreamFormatFromIdentifier(  
    lIdentifier, &pIStreamFormat);
```

## FrameReceived event

### Prototype

```
FrameReceived(IN FireiXFrame Frame)
```

### Comments

This event is fired whenever a frame is received from the camera, after it has been converted to RGB for viewing, but before having been actually sent to the screen (if applicable). It is therefore useful both to perform image processing using the provided FireiXFrame object and manipulation before display.

The supplied FireiXFrame object can be considered valid in the context of the event handler, and any changes made to it (through its methods) are then reflected on the preview screen (if applicable) upon exit from the event handler.

### Syntax

Since event handlers are usually created automatically by the underlying programming environment, it would not be useful to provide sample code here.

## BufferReceived event

### Prototype

```
BufferReceived(IN Variant pBuffer)
```

### *Comments*

This event is fired whenever a frame is received from the camera, before it has been converted to RGB for viewing. It is useful when direct access to the frame buffer being received is necessary, such as when working with Y-MONO-16 formats, since the actual buffer is available.

The supplied `Variant` variable is a regular `SafeArray of Bytes` containing the frame buffer. In a VB6 environment, the `UBound` and `LBound` functions can be used to ascertain the size of the Array (it will be equal to the frame buffer size, in bytes). Each byte in the Array can be accessed directly, using the regular Array accessor.

### *Syntax*

Since event handlers are usually created automatically by the underlying programming environment, it would not be useful to provide sample code here.

## **DeviceRemoved event**

### *Prototype*

`DeviceRemoved()`

### *Comments*

This event is only ever fired once during the lifecycle of the `FireiXCamera` object, if the camera is physically removed from the system.

Since calling any methods or properties of the camera after it has been removed would result in an error, this event notifies that the camera has been unplugged, giving the programmer the chance to handle the situation.

### *Syntax*

Since event handlers are usually created automatically by the underlying programming environment, it would not be useful to provide sample code here.