QCam API

QCam Version: 2.0.13

Document Version: 1.3

Table of Contents:

Applicability

This document applies to the QCam API version 2.0.12. To download the latest version, please visit www.qimaging.com

Notice of Copyright

Trademarks and Proprietary Names

QImaging (Quantitative Imaging Corporation) Address

19535 56th Avenue, Suite 101
Surrey, Canada V3S 6K3
www.qimaging.com

API History

This section describes the set of changes that were made for the specific version of the QCam API mentioned.

QCam 2.0.13

- Added significantly clearer code examples to installation.
- Improved image delay issues with QI-Bolt camera.
- Added .NET-based adapter for QCam.

QCam 2.0.12

- Added qprmFrameBufferLength for changing the size of the QCam internal buffer.
- Fixed bug related to switching from software to hardware triggering and missing a frame.
- Added additional tags to QCapture Suite saved .tiff images.
- Added support for QICommand.
- Improved behaviour of QCam_QueueSettings where under certain circumstances, extra triggers were needed to recover.
- Improved internal frame buffering in QCam resulting in more reliable frame delivery.
- Added missing QCaptureTwain.ds file back into “Plugins” folder.

QCam 2.0.11

- Added support for new camera type qcCameraGoBolt
- Updated QCam version reporting system to work better with 2 digit decimal places.

QCam 2.0.10

- Added support for automatic detection of multicore processor systems for post processing options.

- Optimized libraries to reduce execution overhead.
- Added support for time stamping of frames
- Added support for automatic fire-wire bandwidth management.
- Resolved issues with triggering and abort sequences.
- Memory resources corruption fixed.

QCam 2.0.9

- Added support for new camera type qcCameraRoleraEMC2
- Added expandable settings structure.
- Added Easy EM and gain locked modes.
- Additional memory leaks fixed.

QCam 2.0.8

- WDF driver
- No longer using the Unibrain stack. Can work with either Microsoft or Thesycon stack (use of the Thesycon stack is HIGHLY recommended, because using the Microsoft stack will not achieve maximum throughput from cameras and doesn't support S800 cameras). Stack switching no longer required
- Added support for new camera types, qcCameraRetiga4000DC, qcCameraRetiga2000DC, qcCameraRetigaBlue and qcCameraRetigaGreen.
- Memory leaks fixed.

QCam 2.0.7

- Official release for Retiga EXL
- Fixed minor memory leaks

QCam 2.0.6

- Added support for new camera types, qcCameraRetigaEXL, qcCameraRoleraXRL and qcCameraRetigaSRVL

- Resolved issues with RGB48 images and Go cameras
- Resolved issues with memory leaks when listing, opening and closing cameras.

QCam 2.0.5

- Added support for Windows Vista x86/x64.
- Bayer interpolation through QCam_BayerToRGB() is now accelerated on Intel/AMD machines (under Windows, Mac OS X and Linux).
- Resolved issues when switching FireWire drivers under Windows.
- cameraType item is now filled in inside the QCam_CamListItem structure.

QCam 2.0.4

- Resolved issues with the RGB filter wheel.
- Resolved issue when setting an exposure through QCam_SetParam/QCam_SetParam64 to a value of 0. It now returns the proper error code in this case.
- Resolved issue of Go 21 oversample mode not returning an image correctly from the Photoshop/ImageJ and TWAIN plugins.

QCam 2.0.3

- Added new camera types, qcCameraGo1, qcCameraGo3, qcCameraGo5 and qcCameraGo21.
- Added new parameter, qprmOverSample.
- Added new info parameter, qinfDualChannel.
- Added support for FireWire 800.
- Merged qcamimgfnc.lib into QCamDriver.lib. This means that you longer need to link against both libs – you only need to link to QCamDriver.lib.

QCam 2.0.0

- Added new info parameter, qinfColorWheelSupported.
- Added functions QCam_GetSerialString() and QCam_PostProcessSingleFrame().

- Added Linux support.

QCam 1.90.0

- Added new camera types, qcCameraRoleraMGi, qcCameraReserved1, qcCameraReserved2, and qcCameraOem4
- Added support for clearing modes on frame transfer CCDs and control of active readout port for cameras with multiple readout ports

QCam 1.81.2

- Added new camera type, qcCameraOem3.
- Added support to control shutter delays and to set the EM gain.
- Added support for Asymmetrical Binning

QCam 1.81.0

- Added new camera type, qcCameraRetigaSRV.
- Added support for High Sensitivity Mode, Blackout Mode, Fan Control, and Regulated Cooling.

QCam 1.73.0

- Resolved potential memory leak in QCam_QueueFrame()

QCam 1.72.0

- Added new image type, qfmtRgb24.
- Added new camera type, qcCameraRoleraXR.
- Added new image processing algorithms, QCam_AutoExpose() and QCam_WhiteBalance().
- API documentation has been re-written.
- Changed QCam_QueueFrame() and QCam_GrabFrame() to automatically do post processing on the image if the qprmDoPostProcessing option has been set.

QCam 1.71.0

- Resolved issue with QCam_GrabFrame() not returning an error code. The function now returns an error if there was a problem with the frame.
- The Retiga 2000R now returns a camera type of qcCameraRet2000R.
- The Retiga 4000R now returns a camera type of qcCameraRet4000R.

QCam 1.70.0

- Added new bicubic bayer algorithms, qcBayerBicubic and qcBayerBicubic_Faster for QCam_BayerToRGB().

QCam 1.68.6

- Added additional sensor types under QCam_qcCcd in the QCam API header.
- QCam forwards compatibility (limited).

QCam 1.68.0

- Real Time Viewing Mode for MicroPublisher camera models.
- Normalized CCD Gain, Normalized Intensifier Gain.
- Signed 32 bit integer parameter support.
- Absolute Offset

QCam 1.66.0

- Packetsize and Bandwidth Healing
- Fixed Color Wheel (Now a Sparse Table Parameter)

QCam 1.64.0

- Addition of sparse tables. It is recommended that applications make use of sparse tabling whenever possible. Please review the functions IsSparseParam and IsRangeParam for more information.

QCam 1.62.0

- 64-bit exposure time. The exposure, red exposure and blue exposure used to use 32-bit exposure times. Now all three use 64-bit exposure times.

- Addition of Intensifier Support

QCam 1.50.0

- Initial QCam SDK Documentation.

Introduction

This document is a programmer's guide to the QCam API, which is the programming interface to QImaging's digital cameras.

The QCam API is available for Windows 2000/XP, Mac OS X (10.4.7 or higher), and Linux (kernel version 2.6 or higher).

The different layers of interaction between an application, QCam and the operating system are detailed in the following illustration:

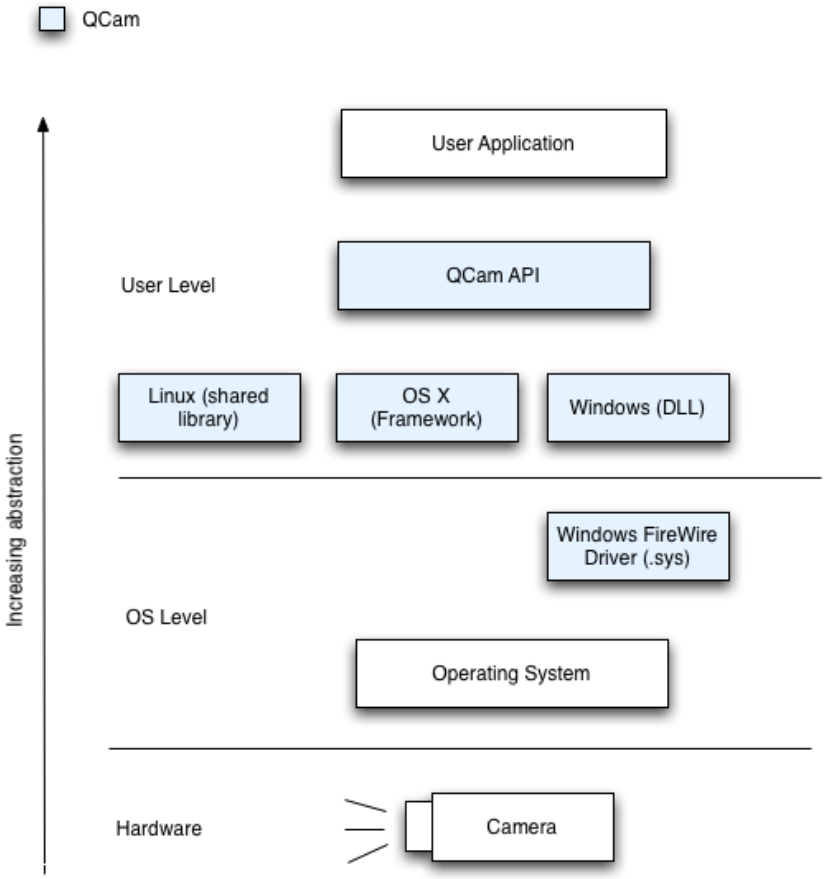


Figure 1 – QCam Layer Stack

The SDK (Software Development Kit) that is provided contains QCam (a framework or dll), the latest driver installer, code examples as well as this document.

Objective

The object of this document is to illustrate to the user how to use the QCam API as well as provide documentation for each available call in the API. The reader of this document should have a good knowledge of C/C++.

Purpose

The purpose of the QCam API is to provide the following:

- Cross-platform interface to control the cameras
- Easy to use function calls
- Provide accessibility to all the camera features
- One common interface that will control any QImaging digital camera

Programming Language

The recommended language to be used is either C or C++. A C/C++ header file is included in the SDK.

You may choose to use a different language, such as Java or C# which will require you to write a small wrapper layer in that language to call into the QCam API using C/C++.

All of the callbacks functions provided to QCam must be written in C. If you are using a different language, you again must write a wrapper to call from the callback function into your language.

Linking

Win32: QCamDriver is a dll that resides in the \WINDOWS\system32 directory. If you are using Visual C++ 6.0, a stub library called QCamDriver.lib is provided to link against. Most other compilers can create a stub library (.lib) from the dll. The QCamDriver dll uses Windows standard calling conventions (ie. __stdcall).

Mac OS X: QCam.framework resides in your /Library/Frameworks folder. It contains the QCamAPI.h header file as well as the QCam library. If you are using Xcode or CodeWarrior, simply include this framework in your project.

Linux: libQCam.so resides in the /usr/local/lib/qimaging directory. There is a symbolic link in/usr/lib that links to it. The headers files are located in /usr/local/include/qimaging.

Sparse / Range Tables

The QCam API uses what's called sparse and range tables for certain functions calls.

A sparse table is a list where only certain values are set. For example, a sparse table for the binning modes supported by a specific camera could contain [1, 2, null, 4, null, null, null, 8] to show that it supports binning by 1,2, 4 or 8.

A range table is a list that contains a range of values, such as 1,2,3,4,5 for the values 1 to 5.

Note: For backwards compatibility with earlier versions of the API, some sparse parameters will continue to support ranges. Requesting a parameter (such as binning) as a range when it should be sparse may result in inaccurate information.

Using the QCam API

The following sections describe how to use the API to do various operations such as finding and opening a camera connection, grabbing an image, getting/setting camera settings, trigger modes and image formats.

Note: The examples shown do not contain error checking. It has been omitted for the sake of clarity.

Loading QCam

The first call you must make is **QCam_LoadDriver()**. This initializes QCam and prepares it for communication with the camera. The last call you must make is **QCam_ReleaseDriver()** to unload the driver and allow it to clean up.

Example 1 – Loading / unloading the driver:

```
QCam_LoadDriver();
...// do some stuff
QCam_ReleaseDriver();
```

Opening a Camera

First, you must get a list of all the available cameras. This can be done with a call to **QCam_ListCameras**(). Once you have a list of cameras, you may open a connection to one with a call to **QCam_OpenCamera**().

Example 2 - Simple example that opens up the first available camera:

```
QCam_CamListItem        list[10];
unsigned long           listLen =
sizeof(list)/sizeof(list[QCam_CamListItem]);

// load the driver
QCam_LoadDriver();

// get a list of the cameras
QCam_ListCameras(list, &listLen);

// listLen is now the number of cameras available.  It may be
```

```
// larger than your QCam_CamListItem array length!

if (( listLen > 0 ) && ( list[0].isOpen == false ))
{
      QCam_Handle       myHandle;

      // Open the first camera in the list.
      QCam_OpenCamera(list[0].cameraId, &myHandle);

      // Do something...
      doSomethingWithTheCamera();

      // Close the camera.  We are done.
      QCam_CloseCamera(myHandle);
}

// close the driver
QCam_ReleaseDriver();
```

The **cameraId** field of the **QCam_CamListItem** structure contains a unique ID for the camera. Many developers use the unique ID to recall previous settings, or associate a meaningful name string with a camera.

Camera Settings

QCam_Settings as of version 2.0.9 has become obsolete and may be deprecated on future releases, therefore its use is not recommended for new designs. **QCam_SettingsEx** along with **QCam_CreateCameraSettingsStruct** and **QCam_ReleaseCameraSettingsStruct** should be used instead.

Modifying Camera Settings

You can read the default state of the camera, read settings from the camera and write settings to the camera.

The camera state is represented by the opaque structure **QCam_SettingsEx**.

Parameters are read and written to the camera using the **QCam_GetParam(), QCam_GetParam64()**, **QCam_SetParam(), QCam_SetParam64()** functions.

The functions **QCam_IsRangeTable()** and **QCam_IsSparseTable()** may be used to determine if range and sparse tables are supported for the parameter. The functions **QCam_GetParamMin()**, **QCam_GetParamMax()** and **QCam_GetParamSparseTable()** are used to get the range and sparse table values.

Note: Some parameters are 32-bit while others are 64-bit (such as exposure time). Ensure that the correct function is used to get/set the parameter.

Here is a list of parameters that use range and sparse tables:

Range Parameter Types	Sparse Parameter Types
Exposure	Cooling
Red Exposure	Readout Speed
Blue Exposure	Shutter State
Gain	Sync B Mode
Offset	Binning (Square, Vertical, Horizontal)
Region of Interest (X, Y, Width, Height)	Trigger Type
Intensifier Gain	Image Format
Trigger Delay	Color Wheel
Normalized Gain / Normalized Gain dB	Camera Mode
Normalized Intensifier Gain / Normalized Intensifier Gain dB	
Absolute Offset	

Table 1 – Parameters and Table Type

Example 3 - Setting the camera exposure time to 100ms.

```
QCam_SettingsEx mySettings;

// Create and allocate the settings structure
QCam_CreateCameraSettingsStruct(&mySettings);

// Initialize the newly created structure
// with the camera default settings.
QCam_InitializeCameraSettings (myHandle, &mySettings);

// Change mySettings: set exposure to 100 ms.
QCam_SetParam64( (QCam_Settings*) &mySettings, qprm64Exposure,
100000000);
```

```
// Send settings to the camera
QCam_SendSettingsToCam(myHandle, (QCam_Settings*) &mySettings);

// release the settings structure when no longer needed
//
QCam_ReleaseCameraSettingsStruct(&mySettings);  // Don't forget this!
```

Example 4 - Using a sparse table to determine the valid binning modes

```
QCam_SettingsEx mySettings;
unsigned long   uTable[32];
unsigned long   uSize = 32;

// Create and allocate the settings structure
QCam_CreateCameraSettingsStruct(&mySettings);

// Initialize the newly created structure
// with the camera default settings.
QCam_InitializeCameraSettings (myHandle, &mySettings);

if (QCam_IsSparseTable( (QCam_Settings*) &mySettings, qprmBinning) ==
qerrSuccess)
{
      // This function will put the number of binning modes
      // into uSize and all of the supported binning modes into
      // uTable.
      QCam_GetParamSparseTable( (QCam_Settings*) &mySettings,
                              qprmBinning,
                              uTable,
                              &uSize);
}
```

The variable *uTable* will contain the binning modes supported by the camera, and *uSize* will be the size of the table.

Example 5 - Using a range table to determine the 64-bit exposure range

```
QCam_SettingsEx   mySettings;
uint64            uMin, uMax;

// Create and allocate the settings structure
QCam_CreateCameraSettingsStruct(&mySettings);

// Initialize the newly created structure
// with the camera default settings.
QCam_InitializeCameraSettings (myHandle, &mySettings);

if (QCam_IsRangeTable64( (QCam_Settings*) &mySettings, qprm64Exposure)
== qerrSuccess)
{
      // This function will get the Minimum and Maximum Exposure
      // times (64-bit)
      QCam_GetParam64Min( (QCam_Settings*) &mySettings, qprm64Exposure,
&uMin);
```

```
      QCam_GetParam64Max( (QCam_Settings*) &mySettings, qprm64Exposure,
&uMax);
}
```

Image Types

There are four image formats that QCam uses. Depending on your application and the capabilities of your camera (mono, colour or rgb-filter) you can decide which image format is appropriate to use.

Monochrome	Used for monochrome sensors, or for getting a monochrome image from a colour sensor in binning mode.
Bayer	Used for colour (bayer) sensors in 1x1 (non-binned) mode.
Colour RGB-Filter	Used to create a colour image from a monochrome sensor. QCam automatically drives the filter to combine R (red), G (green) and B (blue) colour planes into a single colour image.
Raw	Used for OEMs.

There are four different Bayer patterns used with QCam. They are:

qcBayerRGGB	(Red, Green, Green, Blue)
qcBayerGRBG	(Green, Red, Blue, Green)
qcBayerGBRG	(Green, Blue, Red, Green)
qcBayerBGGR	(Blue, Green, Green, Red)

qcBayerRGGB

R	G	R	G	
G	B	G	B	
R	G	R	G	
G	B	G	B	

qcBayerGRBG

G	R	G	R	
B	G	B	G	
G	R	G	R	
B	G	B	G	

Figure 2 - Bayer patterns.

Bayer interpolation (ie: converting RGGB into RGB) is not done automatically by QCam once a frame has been captured. Use the Bayer interpolation function **QCam_BayerToRgb()** that is available in the QCamImgFnc library. The library is distributed as part of the SDK. You can determine which Bayer pattern is used for a frame by looking at the *bayerPattern* field of your *QCam_Frame* structure.

Notes: - The Micropublisher is capable of outputting binned Bayer images.

- The raw image format is a specialized image format. It may change without notice.

Grabbing a Frame

To grab an image from the camera, you first need to allocate a buffer that is large enough to hold all the image data. Next, fill out a *QCam_Frame* structure, and then call a grab function. There are two different ways to get an image, synchronously and asynchronously. The first type blocks until the frame is captured. The second involves using a callback so your application is not blocked. Asynchronous grabs are covered later.

Example 6 – Simple frame grab of a colour image

```
QCam_Frame        origFrame;
QCam_Frame        processedFrame;
unsigned long     sizeInBytes;

// Image size depends on the current region & image format.
QCam_GetInfo(myHandle, qinfImageSize, &sizeInBytes);

// Fill out fields in QCam_Frame structure.
origFrame.pBuffer = new unsigned char[sizeInBytes];
origFrame.bufferSize = sizeInBytes;

// Do synchronous image grab.
// This function will not return until the frame has been captured
QCam_GrabFrame(myHandle, &origFrame);

// Process the data.
QCam_BayerToRgb(qcBayerInterpBiCubic_Faster, &origFrame,
                  &processedFrame);
```

```
// Do something with the frame
doSomethingWithTheFrame(processedFrame);

delete []origFrame.pBuffer;
delete []processedFrame.pBuffer;
```

Gain and Offset (DEPRECATED)

Gain is now controlled with the Normalized Gain parameter *qprmNormalizedGain* and offset values are controlled with Absolute Offset parameter *qprmS32AbsoluteOffset*. Cameras manufactured after March 1, 2004 will no longer respond to parameter queries *qprmGain* and *qprmOffset*. For backward compatibility, cameras manufactured before Mar 1, 2004 will continue to work with parameters *qprmGain* and *qprmOffset*. For more information, see the following sections on Normalized Gain and Absolute Offset.

Normalized Gain and Normalized Gain dB

Normalized gain controls the factor by which the analog signal from the CCD is multiplied before digitization. Normalized gain operates in much the same way that its predecessor gain has, but the representation of normalized gain is more meaningful. The default, factory optimized, normalized gain is always one. A normalized gain value represents a multiple of the default intensity. The default gain value corresponds to the gain value that results in the camera's highest achievable dynamic range.

Two parameters can be used to change the normalized gain value: *qprmNormalizedGain* and *qprmS32NormalizedGaindB*. Normalized gain dB is a decibel representation of normalized gain where 0 dB always corresponds to the default, factory optimized gain of 1.

Wherever a camera's gain control is presented to an end-user, QImaging recommends the use of *qprmNormalizedGain* represented as a floating point number rounded to the number of significant figures reported by *qinfNormGainSigFigs*. If *qprmNormalizedGain* is represented with a slider control, QImaging recommends that its mapping be logarithmic. This type of mapping can be seen in the QCapture software that is included with the camera.

qprmS32NormalizedGaindB is intended for use where QImaging cameras are integrated into automated imaging systems where the end-user does not typically interact with the gain control. *qprmS32NormalizedGaindB* provides an interface to the gain control where the resolution is constant across the range of possible gain values as reported by

qinfNormGaindBRes. The constant resolution can simplify machine control of the camera gain calculations, which are often performed in dB.

Note: Parameters *qprmNormalizedGain* and *qprmS32NormalizedGaindB* are expressed in micro units.

Normalized Intensifier Gain and Normalized Intensifier Gain dB

Normalized intensifier gain controls the gain of the camera's intensifier. In the case where the camera employs an intensifier tube (an example is the Intensified Retiga), the intensifier gain corresponds to the optical gain of the intensifier tube. Normalized intensifier gain operates in much the same way that its predecessor intensifier gain has, but the representation of normalized intensifier gain is more meaningful. The default intensifier gain corresponds to a value that will produce quality images in most typical situations.

Two parameters can be used to change the normalized intensifier gain value: *qprm64NormIntensGain* and *qprmNormIntensGaindB*. Normalized intensifier gain dB is a decibel representation of normalized intensifier gain where 0 dB always corresponds to a gain of 1.

The recommendations made for use of normalized gain also apply to normalized intensifier gain.

Note: Parameters *qprm64NormIntensGain* and *qprmNormIntensGaindB* are expressed in micro units.

Absolute Offset

Absolute offset is much the same as its predecessor offset except that absolute offset is mapped such that a value of 0 always corresponds to a digitized black level of 0. Positive and negative absolute offset values correspond to positive and negative digitized black levels, respectively. The default, factory optimized, absolute offset value is always zero. This value corresponds to the offset value that results in the camera's highest achievable dynamic range.

Because absolute offset is more meaningful and gives consistent black level control, QImaging recommends that it be used in all cases where the black level of the camera is manipulated.

Binning

To change the binning on a camera that supports it, use the *qprmBinning* parameter with the **QCam_SendSettingsToCam()** call.

The maximum size for the binning mode chosen is not an even division of the 1x1 maximum image size. When you call **QCam_SendSettingsToCam()** or **QCam_PreflightSettings()**, the ROI (Region of Interest) will be adjusted to fit the binning mode.

When switching from a higher binning mode (ex: 4x4) to a lower binning mode (ex: 1x1), you must adjust your ROI. This is because the lower the binning mode is, the larger the ROI will be. You can set the ROI to its maximum width / height to resolve this issue since the ROI will be adjusted to fit the binning mode.

Example 7 – Settting ROI to maximum width / height

```
unsigned long maxWidth;
unsigned long maxHeight;

QCam_GetInfo(myHandle, qinfCcdWidth, &maxWidth);
QCam_GetInfo(myHandle, qinfCcdHeight, &maxHeight);

QCam_SetParam( (QCam_Settings*) &mySettings, qprmRoiX, 0);
QCam_SetParam( (QCam_Settings*) &mySettings, qprmRoiY, 0);
QCam_SetParam( (QCam_Settings*) &mySettings, qprmRoiWidth, maxWidth);
QCam_SetParam( (QCam_Settings*) &mySettings, qprmRoiHeight, maxHeight);
```

Note: The Micropublisher is capable or vertical binning with the *qprmVerticalBinning* parameter. It is a special form or vertical binning that maintains colour information. When using this binning mode, the application must provide horizontal software binning or decimation to restore a normal aspect ratio.

Region of Interest

To change the Region of Interest (ROI), use the *qprmRoiX*, *qprmRoiY* , *qprmRoiWidth* and *qprmRoiHeight* parameters with the **QCam_SendSettingsToCam()** call.

The ROI can be set on 1-pixel boundaries, down to 1x1 pixels. When you are running in a binning mode, your region and image size will be specified in super-pixels rather than CCD pixels.

Note: On cameras purchased before September 2002 the following limitations apply:

- ROI boundaries occur on multiplies of 16
- The minimum ROI is 16x16 (because of the previous rule)
  - QCam_SendSettingsToCam() and QCam_PreflightSettings() will automatically adjust the ROI to fit the 16 pixel boundary

Readout Speed

To change the readout speed, use the *qprmReadoutSpeed* parameter with the **QCam_SendSettingsToCam()** call.

The maximum readout speed is determined by three factors: **a)** the camera hardware **b)** the amount of available FireWire bandwidth **c)** the operating system being used. If a readout speed is requested that is not supported, QCam will lower the speed to the next appropriate level.

When changing a camera setting, such as bit depth, you may want to reset the readout speed.

Example 8 – Changing the readout speed to 20Mhz

```
QCam_SetParam( (QCam_Settings*) &mySettings, qprmReadoutSpeed,
qcReadout20M);
```

Queuing Frames (Asynchronous)

A function called **QCam_QueueFrame()** is available to grab images asynchronously. This function allows many frames to be queued up at a time. When an image arrives from the camera and a frame is complete, your application will receive a callback. The completed frame is then removed from the queue, and if there is one available, the next frame will take its place.

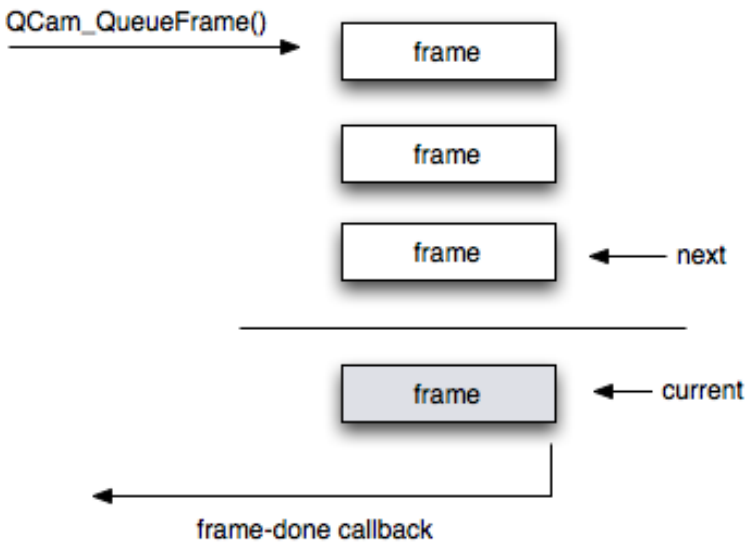


Figure 3 – QCam frame queue

Using **QCam_QueueFrame()** has the following advantages over a synchronous grab:

- The calling thread is not blocked
- Queued frames are easily aborted
- Software trigger only works with queued frames
- Frames can be captured at full speed

To capture frames at full speed, at least two frames should be continuously queued. This is due to the fact that the newer cameras can simultaneously readout and expose. Without two frames queued up, this can be a problem if the exposure time is less than the readout time.

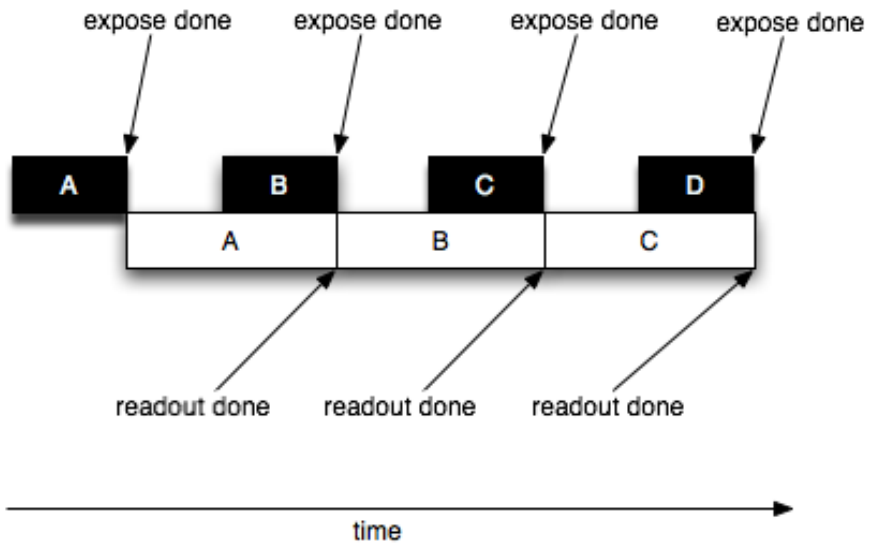


Figure 5 – Simultaneous readout and expose

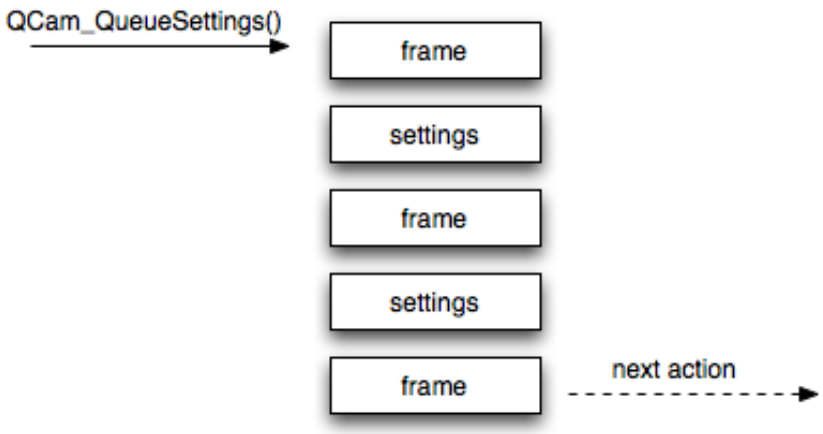
In Figure 5, if a new frame is not immediately available when the current frame finishes, the next image will be lost.

Image processing should be done a separate thread and not during the expose-done callback. An example of how this would be used is in an application with a graphical user interface (GUI). The application might control the camera and queue frames on their main thread, and perform lengthy processing on a separate thread.

Queuing Settings (Asynchronous)

To queue changes to the camera settings, the function **QCam_QueueSettings()** is available. The settings are placed on the same queue as frames queued up by **QCam_QueueFrame()**. Actions are guaranteed to occur in the order they were queued in. To clear the queue, call **QCam_Abort()**.

As with **QCam_QueueFrame()**, your application will receive a callback when the settings have been changed.



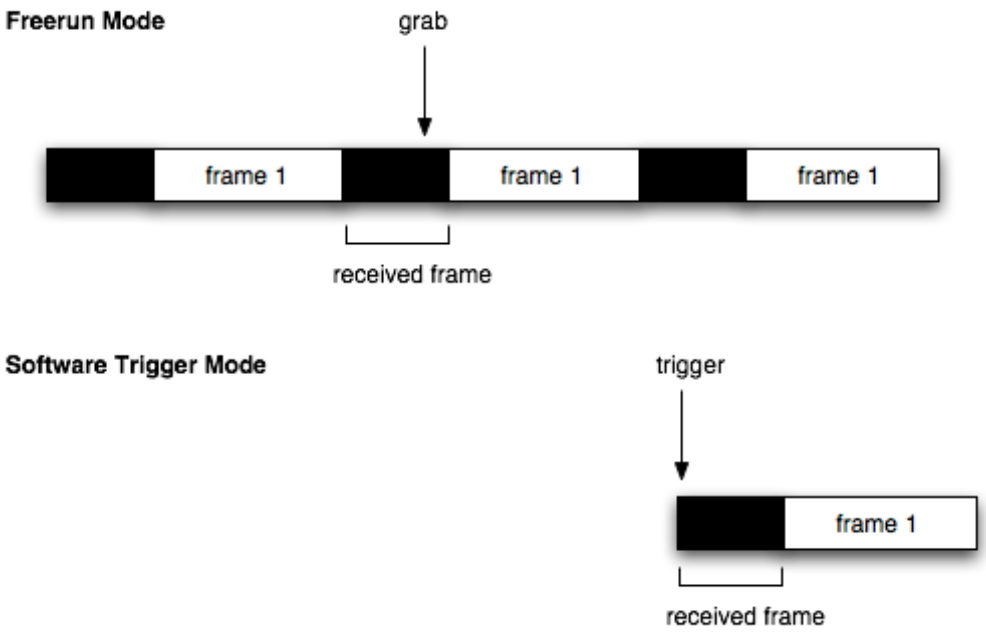
QCam_QueueSettings() can be very useful in particular instances, although most applications will use the simpler, and synchronous **QCam_SendSettingsToCam()**.

Trigger Modes

The trigger modes in QCam consist of three categories:

Freerun	The camera is in a continuous expose-readout loop. On newer cameras, the next exposure will start while the last exposure is reading out.
Hardware Trigger	The exposure starts on an external hardware trigger signal.
Software Trigger	The exposure starts on demand, when **QCam_Trigger()** is called.

When synchronizing images with an external device, such as a motorized stage, use the hardware or software trigger mode. Using the freerun mode could be unreliable, since the exposure might start before the grab function is called (see Figure 7).



The latency of the software trigger mode is approximately 2-3 ms.

Note: On cameras purchased before September 2002, the latency can be approximately 60-80 ms.

Real Time Viewing Mode

Real Time Viewing (RTV) mode is a non-interlaced mode available on MicroPublisher models. It trades higher frame rates for a small reduction in image quality. To enable this feature, set the parameter *qprmCameraMode to qmdRealTimeViewing*. To return to the Standard Mode, use *qmdStandard*. The formatting operations described below can also be performed automatically by using the *qprmDoPostProcessing* mode described in the next section.

In this mode, binning is forced to 1x1.

Note: The pixel intensity is twice that of Standard Mode with 1x1 binning.

3.3 Megapixel Micropublisher

In RTV, the size of the images obtained from the camera will be 1/6 of the height of *qmdStandard*. The width will remain unchanged. Decimating 2/3 pixels

horizontally and interpolating every second row vertically will result in an image that is 682x512.

5.0 Megapixel Micropublisher

In RTV, the size of the images obtained from the camera will be 1/8 of the height of *qmdStandard*. The width will remain unchanged. Decimating 3/4 pixels horizontally and interpolating every second row vertically will result in an image that is 640x480.

Post Processing

Post processing is available to simplify collecting color images from the raw formats captured by Bayer colour cameras. The main use for post processing is automatically interpolating qfmtBgrx32 or qfmtRgb48 color images out of their respective raw qfmtBayer8 and qfmtBayer16 formats. Post processing will also perform additional processing on raw images to simplify the collection of color images on MicroPublisher cameras, including performing horizontal software binning and processing of RTV frames. Finally enabling post processing allows you to make use of the White Balance call to have QCam white balance the collected images based on a given region of interest.

To enable post processing *qprmDoPostProcessing* must be set to 1. Additionally *qprmPostProcessImageFormat* should be set to the format of the desired output frame. The buffers for that output frame must be allocated using the appropriate image size. The best way to find out the image size that the post processed frame will be is to query the qinfPostProcessImageSize parameter using the QCam_GetInfo call. Below is an example on how to set up post processing.

```
//turn on post processing first
QCam_SetParam( (QCam_Settings*) &mySettings, qprmDoPostProcessing, 1);
//set up the post processing image format to qfmtBgrx32
QCam_SetParam( (QCam_Settings*)  &mySettings,
qprmPostProcessImageFormat,qfmtBgrx32);
//set up the bayer interpolation algorithm
```

```
QCam_SetParam( (QCam_Settings*) &mySettings,
qprmPostProcessBayerAlgorithm, qcBayerInterpAvg4);

//allocate the appropriate memory for the buffers
QCam_GetInfo(qinfPostProcessImageSize , processedSize);

frame.pBuffer = new byte[ processedSize ];
frame.bufferSize = processedSize;
```

After setting up post processing, the appropriately allocated frames can be submitted to the QCam_QueueFrame call or to the QCam_GrabFrame call.

Image Formats

The following are the image formats that are defined by QCam. Some cameras may not support all of the formats.

Note: For the 16-bit formats, the data from the camera may have been 10-bit or 12-bit.

Raw:

qfmtRaw8	Raw 8-bit output from the camera. This format is not supported.
qfmtRaw16	Raw 16-bit output from the camera. This format is not supported.

Mono:

qfmtMono8	Monochrome 8-bit data where each byte represents a pixel. On colour CCD cameras, this format is only supported when binning 2x2 or more.
qfmtMono16	Same as above, but 16-bit.

Bayer:

qfmtBayer8	Raw 8-bit Bayer image from the camera. The Bayer alignment can change, so be sure to check the *bayerPattern* field of the *QCam_Frame* structure. This format is only supported on colour CCD cameras.

qfmtBayer16 Same as above, but 16-bit.

RGB Filter*:*

qfmtRgbPlane8 Red, green and blue planes in monochrome format, one following another.

qfmtRgbPlane16 Same as above, but 16-bit.

Other:

qfmtBgr24 An RGB image where each pixel is 3 bytes. When viewed as an array, the order is as follows: blue, green, red.

qfmtRgb24 An RGB image where each pixel is 3 bytes. When viewed as an array, the order is red, green and blue. This is a common format for images. Note: It does not contain an alpha channel.

qfmtBgrx32 An RGB image where each pixel is 4 bytes. When viewed as an array, the order is as follows: blue, green, red, and the last byte is undefined. It is a common format for Windows bitmaps.

qfmtXrgb32 An RGB image where each pixel is 4 bytes. When viewed as an array, the order is as follows: first byte is undefined, red, green, and blue. It is a common format for QuickDraw bitmaps.

qfmtRgb48 An RGB image where each pixel is 6 bytes. In the array, it goes as follows: red, green, blue (each colour is 2 bytes long).

Camera State Parameters

The following are the parameters to control the state of the camera. They are used in conjunction with **QCam_SetParam()** and **QCam_SendSettingsToCam()**. Some cameras may not support all of the parameters.

qprmBinning Binning mode for cameras with square binning (ex: 1x1 or 4x4)

qprmCameraMode	Micropublisher only – Controls the fundamental operation of the camera's CCD sensor.
qprmCoolerActive	Enable (non-zero) or disable (zero) the thermoelectric cooler.
qprmColorWheel	The colour of the RGB filter. See *QCam_qcWheelColor* in QCamAPI.h for the defined constants.
qprmGain	Deprecated. Use *qprmNormalizedGain*.
qprmHorizontalBinning	Horizontal binning mode for cameras with 2-axis binning control.
qprmImageFormat	The image format. See *QCam_qcImageFormat* in QCamAPI.h for the defined constants.
qprmIntensifierGain	Deprecated. Use *qprm64NormIntensGain*.
qprmNormalizedGain	Normalized signal gain. This is the analog gain before the Digital-to-Analog Converter (DAC). The default gain is always 1 and the gain values represent a multiple of pixel intensity form the default. Expressed in micro units.
qprmNormIntensGaindB	Same as *qprm64NormIntensGain,* but in decibels. Expressed in micro units.
qprmOffset	Deprecated. Use *qprmS32AbsoluteOffset*.
qprmReadoutSpeed	The readout speed of the CCD. See *QCam_ qcReadoutSpeed* in QCamAPI.h for the defined constants.
qprmRoiHeight	The height of the ROI, in pixels.
qprmRoiWidth	The width of the ROI, in pixels.
qprmRoiX	The X-axis value for the upper-left hand corner of the ROI, in pixels. Pixels binned together count as a single pixel.

qprmRoiY	The Y-axis value for the upper-left hand corner of the ROI, in pixels.
qprmS32AbsoluteOffset	The signal offset. This is the analog offset into the DAC. The default offset is 0.
qprmS32NormalizedGaindB	A decibel representation of *qprmNormalizedGain*. Expressed in micro units.
qprmShutterState	The position / mode of the shutter. See *QCam_ qcShutterControl* in QCamAPI.h for the defined constants.
qprmSyncB	Specifies what the Sync B (AuxSync2) signal represents. This is only available on specific cameras and may either be Exposure Mask or Trigger Mask. See the Trigger Circuit Application Note for more information.
qprmTriggerDelay	The trigger delay in nanoseconds.
qprmTriggerType	The exposure trigger type. See *QCam_ qcTriggerType* in QCamAPI.h for the defined constants.
qprmVerticalBinning	Vertical binning mode for cameras with 2-axis binning control. This parameter is now a range and sparse table.
qprm64Exposure	The exposure time in nanoseconds. For RGB colour filter images, this is the green exposure.
qprm64ExposureBlue	For RGB colour filter images. This is the blue exposure time in nanoseconds.
qprm64ExposureRed	For RGB colour filter images. This is the red exposure time in nanoseconds.
qprm64NormIntensGain	Intensified Cameras Only - Normalized Intensifier Gain. This is the amount by which the intensifier

	state will increase light intensity. Expressed in micro units.
qprmDoPostProcessing	For enabling the post processing pipeline. It specifies that all images queued or grabbed need to be bayer interpolated, binned or color balanced. Must be 0 to do no processing and 1 to do processing.
qprmPostProcessGainRed	For setting the red scale value used in color balancing the bayer cameras.
qprmPostProcessGainGreen	For setting the green scale value used in color balancing the bayer cameras.
qprmPostProcessGainBlue	For setting the blue scale value used in color balancing the bayer cameras.
qprmPostProcessBayerAlgorithm	For specifying the Bayer Interpolation algorinthm used in the post processing of the image.
qprmPostProcessImageFormat	Specifies which image format the post processed image will be in (ex: qfmtXrgb32).
qprmS32RegulatedCoolingTemp	For setting the temperature of the cooler on cameras that support regulated cooling.
qprmFan	For setting the speed of the fan. Use one of the QCam_qcFanSpeed enums: qcOneQuarterSpeed, qcHalfSpeed, qcThreeQuarterSpeed, or qcFullSpeed.
qprmBlackoutMode	For setting blackout mode which turns all the light emitting areas of the camera on / off. (ex: the OLED)
qprmHighSensitivityMode	For setting the high sensitivity mode.
qprmEMGain	For setting the EM gain on a camera.

qprmCloseDelay	For setting the time at which shutter will close before the falling edge of the exposure signal.
qprmOpenDelay	For setting the time at which shutter will open after the rising edge of the exposure signal.
qprmCCDClearingMode	For setting the clearing mode used on frame transfer CCDs
qprmOverSample	For settings oversample mode. Currently the only camera that supports this is the Go 21.
qprmEasyEmMode	For enabling/disabling EasyEM mode which locks the EM gain on a camera to its 100X gain setting when enabled.
qprmLockedGainMode	For enabling/disabling Gain Locked mode which locks the Normalized Gain on a camera to its 2 electrons per ADU when enabled.
qprmFrameBufferLength	For setting the maximum internal frame buffer length. If the user does not process the incoming frames fast enough or the CPU load is too high, the new frames are stored in an internal buffer. If the number of frames in the buffer exceeds the *qprmFrameBufferLength* the most recent frame replaces the latest frame in the buffer.

Camera Information Parameters

The following parameters are used to query information about the camera. Some cameras may not support all of the parameters.

qinfBitDepth	The maximum bit depth available.
qinfCameraType	The camera model. See *QCam_ qcCameraType* in QCamAPI.h for the defined constants.
qinfCcd	The CCD model. See *QCam_ qcCcd* in QCamAPI.h for the defined constants.
qinfCcdHeight	The maximum image height available from the camera.
qinfCcdWidth	The maximum image width available from the camera.
qinfCcdType	The CCD type (ie: monochrome or Bayer). See *QCam_ qcCcdType* in QCamAPI.h for the defined constants.
qinfCooled	This is a non-zero value if the camera has cooling available. It does not indicate if cooling is turned on/off.
qinfExposureRes	The exposure time resolution in nanoseconds.
qinfFirmwareBuild	The firmware build version of the camera.
qinfFirmwareVersion	The firmware software version of the camera.
qinfHardwareVersion	The hardware version of the camera.
qinfImageHeight	The current image height. This is the same as *qprmRoiHeight*.
qinfImageWidth	The current image width. This is the same as *qprmRoiWidth*.

qinfImageSize	The image size for the current format, in bytes.
qinfIntensifierModel	The intensifier tube model.
qinfIsModelB	Returns 1 if the camera is a Model-B.
qinfNormGaindBRes	The normalized gain dB resolution. The values will be rounded to the nearest resolution. Expressed in micro units.
qinfNormGainSigFigs	The number of significant figures in the normalized gain. Normalized gain will be rounded to the nearest significant figure.
qinfNormITGaindBRes	The normalized intensifier gain dB resolution. Same as *qinfNormGaindBRes* but for intensified cameras.
qinfNormITGainSigFigs	The number of significant figures in the normalized intensified gain. Same as *qinfNormGainSigFigs* but for intensified cameras.
qinfSerialNumber	The serial number of the camera. If the value is zero, it indicates the serial number was not set.
qinfStreamVersion	The streaming version.
qinfTriggerDelayRes	The trigger delay resolution, in nanoseconds.
qinfUniqueId	The unique ID of the camera. This is the same as in *QCam_CamListItem*.
qinfRegulatedCooling	Returns 1 if the camera supports regulated cooling, 0 otherwise.
qinfRegulatedCoolingLock	Returns 1 if the camera has reached temperature lock, 0 otherwise.
qinfFanControl	Returns 1 if the camera supports changing the fan speed, 0 otherwise.
qinfHighSensitivityMode	Returns 1 if the camera supports High Sensitivity Mode, 0 otherwise.

qinfBlackoutMode	Returns 1 if the camera supports blackout mode, 0 otherwise.
qinfPostProcessImageSize	Returns the size in bytes of the post processed image.
qinfAsymetricalBinning	Returns TRUE if asymmetrical binning is supported.
qinfColorWheelSupported	Returns 1 if the color filter wheel is supported, 0 otherwise.
qinfEMGain	Returns TRUE if EM Gain control is supported.
qinfOpenDelay	Returns TRUE if open delay control is supported.
qinfCloseDelay	Returns TRUE if close delay control is supported.
qinfColorWheelSupported	Returns TRUE if the rgb color wheel is supported.
qinfDualChannel	Returns TRUE if dual channel mode is supported.
qinfEasyEmModeSupported	Returns TRUE if EasyEM mode is supported.
qinfLockedGainModeSupported	Returns TRUE if Locked Gain Mode is supported.

Result Codes

The following are the result codes that can be returned by functions in the QCam API.

qerrSuccess	Success.
qerrNotSupported	The function called is not supported for the current camera.
qerrInvalidValue	A parameter value passed is invalid.
qerrBadSettings	The QCam_SettingsEx structure passed in is invalid.
qerrNoUserDriver	No driver was found.
qerrNoFirewireDriver	No FireWire device driver was found.
qerrDriverConnection	The connection to the driver is broken.
qerrDriverAlreadyLoaded	The driver has already been loaded with a call to QCam_LoadDriver()
qerrDriverNotLoaded	The driver has not been loaded yet. Call QCam_LoadDriver() to resolve this error.
qerrInvalidHandle	The QCam_Handle used is invalid.
qerrUnknownCamera	The camera is unknown to the version of QCam being used.
qerrInvalidCameraId	There is no camera connected with the id used.
qerrNoMoreConnections	Deprecated. Not used.
qerrHardwareFault	There is a problem with the hardware.
qerrFirewireFault	There is a problem with the FireWire communication.
qerrCameraFault	There is a problem with the camera.
qerrDriverFault	There is a problem inside the driver.

qerrInvalidFrameIndex	The index passed in is invalid.
qerrBufferTooSmall	The frame buffer (pBuffer) of the QCam_Frame structure is too small to hold the entire image.
qerrOutOfMemory	QCam is out of memory.
qerrOutOfSharedMemory	QCam is out of shared memory.
qerrBusy	QCam is currently busy with an operation. This would likely happen in a multi-threaded application.
qerrQueueFull	The queue is full. No more items may be queued.
qerrCancelled	An operation inside QCam was cancelled.
qerrNotStreaming	The function requires that streaming is turned on before being called.
qerrLostSync	Frame synchronization has been lost, so this frame is invalid. This can be caused if QCam does not have enough processing time.
qerrBlackFill	The frame is missing some data which has been filled in with black. This can be caused if the FireWire bus has been saturated with traffic from other devices.
qerrFirewireOverflow	Streaming needs to be restarted.
qerrUnplugged	The camera has been unplugged or turned off.
qerrAccessDenied	The camera is already open.
qerrStreamFault	There is not enough bandwidth to allocate a new stream.
qerrQCamUpdateNeeded	The current QCam driver needs to be updated to work correctly with the camera.
qerrRoiTooSmall	The ROI passed in is too small.

QCam API Functions

QCam_Abort

Aborts any pending frame grabs as well as clearing the frame and settings queues.

```
extern QCam_Err QCAMAPI QCam_Abort
(
    QCam_Handle      handle
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

function result
A result code.

Discussion

Once this call has been made, the callbacks for QCam_QueueFrame() and QCam_QueueSettings() will no longer be called. QCam_Abort() may not be called from these callbacks.

Header: QCamAPI.h

QCam_CloseCamera

Close the connection to a camera.

```
extern QCam_Err QCAMAPI QCam_CloseCamera
(
    QCam_Handle      handle
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

function result
A result code.

Header: QCamAPI.h

QCam_GetCameraModelString

Get the model of the camera as a string.

```
extern QCam_Err QCAMAPI QCam_GetCameraModelString
(
    QCam_Handle          handle,
    char                 *string,
    unsigned long        size
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `string`
The string buffer to copy the camera model string into

➔ `size`
The length of the string buffer.

function result
A result code.

Discussion

This function will return “Unknown” if the camera requires a newer version of the driver.

Header: QCamAPI.h

QCam_GetInfo

Get information from the camera based on the parameter based in.

```
extern QCam_Err QCAMAPI QCam_GetInfo
(
    QCam_Handle      handle,
    QCam_Info        parameter,
    unsigned long    *pValue
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `parameter`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the value that was retrieved from the camera.

function result
A result code.

Discussion

This function returns a camera value based on the parameter based in. An example would be *qinfCcdWidth* which would return the width of the CCD sensor in the camera.

Some parameters are not supported on all cameras. See the section Camera Information Parameters for a list of the different values available.

Header: QCamAPI.h

QCam_GetParam

Read a parameter from the opaque QCam_SettingsEx structure.

```
extern QCam_Err QCAMAPI QCam_GetParam
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    unsigned long        *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the value read. It is always an unsigned 32-bit value.

function result
A result code.

Discussion

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParam64

Read a 64-bit parameter from the opaque QCam_SettingsEx structure.

```
extern QCam_Err QCAMAPI QCam_GetParam64
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    UNSIGNED64           *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the value read. It is always an unsigned 64-bit value.

function result
A result code.

Discussion

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParam64Max

Get the maximum value of a 64-bit parameter.

```
extern QCam_Err QCAMAPI QCam_GetParam64Max
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    UNSIGNED64           *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the maximum value.

function result
A result code.

Discussion

This is the maximum value of the parameter for the state represented by the QCam_SettingsEx structure passed in.

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParam64Min

Get the minimum value of a 64-bit parameter.

```
extern QCam_Err QCAMAPI QCam_GetParam64Min
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    UNSIGNED64           *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the minimum value.

function result
A result code.

Discussion

This is the minimum value of the parameter for the state represented by the QCam_SettingsEx structure passed in.

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParamMax

Get the maximum value of a parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamMax
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    unsigned long        *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the maximum value.

function result
A result code.

Discussion

This is the maximum value of the parameter for the state represented by the QCam_SettingsEx structure passed in.

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParamMin

Get the minimum value of a parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamMin
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    unsigned long        *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the minimum value.

function result
A result code.

Discussion

This is the minimum value of the parameter for the state represented by the QCam_SettingsEx structure passed in.

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParamS32

Read a signed 32-bit parameter from the opaque QCam_SettingsEx structure.

```
extern QCam_Err QCAMAPI QCam_GetParamS32
(
    QCam_Settings const  *pSettings,
    QCam_ParamS32        paramKey,
    signed long          *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the value read in. It is always a signed 32-bit value.

function result
A result code.

Discussion

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParamS32Max

Get the maximum value of a signed 32-bit parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamS32Max
(
    QCam_Settings const  *pSettings,
    QCam_ParamS32        paramKey,
    signed long          *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the maximum value.

function result
A result code.

Discussion

This is the maximum value of the parameter for the state represented by the QCam_SettingsEx structure passed in.

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParamS32Min

Get the minimum value of a signed 32-bit parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamS32Min
(
    QCam_Settings const  *pSettings,
    QCam_ParamS32        paramKey,
    signed long          *pValue
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which information the function will return.

➔ `pValue`
A pointer to the minimum value.

function result
A result code.

Discussion

This is the minimum value of the parameter for the state represented by the QCam_SettingsEx structure passed in.

If the result code is *qerrBadSettings*, be sure to check the following:

- The QCam_SettingsEx structure has been created by calling QCam_CreateSettingsStruct()
- The QCam_SettingsEx structure has been initialized by calling QCam_InitializeCameraSettingsStruct(), QCam_ReadSettingsFromCam(), QCam_ReadDefaultSettings() or QCam_TranslateSettings().

Header: QCamAPI.h

QCam_GetParamSparseTable

Get the sparse table of a parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamSparseTable
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey,
    unsigned long        *pSparseTable,
    int                  *uSize
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read the sparse table from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which table to return.

← `pSparseTable`
A pointer to the sparse table.

←➔ `uSize`
On input, it is the size of the table passed in.

On output, it is the number of elements filled into the table.

function result
A result code.

Discussion

Some parameters do not use sparse tables. Use the call QCam_IsSparseTable() to determine if a sparse table is appropriate for the particular parameter.

Header: QCamAPI.h

QCam_GetParamSparseTable64

Get the sparse table of a 64-bit parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamSparseTable64
(
    QCam_Settings const  *pSettings,
    QCam_Param64         paramKey,
    UNSIGNED64           *pSparseTable,
    int                  *uSize
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_Settings structure to read the sparse table from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which table to return.

➔ `pSparseTable`
A pointer to the sparse table.

➔➔ `uSize`
On input, it is the size of the table passed in.

On output, it is the number of elements filled into the table.

function result
A result code.

Discussion

Some parameters do not use sparse tables. Use the call QCam_IsSparseTable() to determine if a sparse table is appropriate for the particular parameter.

Header: QCamAPI.h

QCam_GetParamSparseTableS32

Get the sparse table of a signed 32-bit parameter.

```
extern QCam_Err QCAMAPI QCam_GetParamSparseTableS32
(
    QCam_Settings const  *pSettings,
    QCam_ParamS32        paramKey,
    signed long          *pSparseTable,
    int                  *uSize
);
```

Parameter Descriptions

➔ `pSettings`
The QCam_SettingsEx structure to read the sparse table from (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The parameter that determines which table to return.

➔ `pSparseTable`
A pointer to the sparse table.

➔➔ `uSize`
On input, it is the size of the table passed in.

On output, it is the number of elements filled into the table.

function result
A result code.

Discussion

Some parameters do not use sparse tables. Use the call QCam_IsSparseTable() to determine if a sparse table is appropriate for the particular parameter.

Header: QCamAPI.h

QCam_GetSerialString

Get the serial string of the camera.

```
extern QCam_Err QCAMAPI QCam_GetSerialString
(
    QCam_Handle          handle,
    char                 *string,
    unsigned long        size
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `string`
The string buffer to copy the serial string into

➔ `size`
The length of the string buffer.

function result
A result code.

Discussion

This function is not supported by cameras purchased before September 2002.

Header: QCamAPI.h

QCam_GrabFrame

Grab one frame from the camera.

```
extern QCam_Err QCAMAPI QCam_GrabFrame
(
    QCam_Handle          handle,
    QCam_Frame           *pFrame
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔➔ `pFrame`
On input, a *QCam_Frame* structure, where the *pBuffer* and *bufferSize* fields have been set to a buffer allocated by the caller.

On output, the image is written into the *pBuffer* field of the *QCam_Frame*.

function result
A result code.

Discussion

The frame and settings queues must be clear before this function is called. If you have set the *qprmDoPostProcessing* parameter then QCam_GrabFrame() will return a QCam_Frame that has been colour balanced, binned and Bayer interpolated.

The following is a list of common result codes returned by this function and a description of why each happened:

qerrInvalidHandle	The camera handle used is not valid.
qerrBufferTooSmall	The frame buffer used is too small to hold the entire image.
qerrBusy	There are pending operations on the queue. It must be cleared before calling this function. See QCam_QueueFrame() and

	QCam_Abort().
qerrLostSync	Something on the FireWire bus has resulted in the frame being out of sync. This frame cannot be used. Isochronous data transmission over FireWire is not guaranteed.
qerrBlackFill	Something on the FireWire bus has resulted in some missing image data. The missing data is filled in with black. Isochronous data transmission over FireWire is not guaranteed.

Header: QCamAPI.h

QCam_IsRangeTable

Determine if a 32-bit parameter is supported as a range table.

```
extern QCam_Err QCAMAPI QCam_IsRangeTable
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey
);
```

Parameter Descriptions

➔ `pSettings`
A QCam_Settings structure to read (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The 32-bit parameter to check.

function result
qerrSuccess if the parameter is a valid range table.

qerrNotSupported if the paramter is not supported as a range table.

Discussion

If the parameter is supported as a range table, then it should have minimum and maximum values.

This function should be used to verify that a parameter is supported as a range table before trying to retrieve it as a range table.

Note: Althrough some parameters that are unsupported as range table still report a range (via QCam_GetParamMin() or QCam_GetParamMax()), it is not recommended that they are used. They are supported for backwards compatibility only.

Header: QCamAPI.h

QCam_IsRangeTable64

Determine if a 64-bit parameter is supported as a range table.

```
extern QCam_Err QCAMAPI QCam_IsRangeTable64
(
    QCam_Settings const  *pSettings,
    QCam_Param64         paramKey
);
```

Parameter Descriptions

➔ `pSettings`
A QCam_Settings structure to read (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The 64-bit parameter to check.

function result
qerrSuccess if the parameter is a valid range table.

qerrNotSupported if the paramter is not supported as a range table.

Discussion

If the parameter is supported as a range table, then it should have minimum and maximum values.

This function should be used to verify that a parameter is supported as a range table before trying to retrieve it as a range table.

Note: Althrough some parameters that are unsupported as range table still report a range (via QCam_GetParam64Min() or QCam_GetParam64Max()), it is not recommended that they are used. They are supported for backwards compatibility only.

Header: QCamAPI.h

QCam_IsRangeTableS32

Determine if a signed 32-bit parameter is supported as a range table.

```
extern QCam_Err QCAMAPI QCam_IsRangeTableS32
(
    QCam_Settings const  *pSettings,
    QCam_ParamS32        paramKey
);
```

Parameter Descriptions

➔ `pSettings`
A QCam_Settings structure to read (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The signed 32-bit parameter to check.

function result
qerrSuccess if the parameter is a valid range table.

qerrNotSupported if the paramter is not supported as a range table.

Discussion

If the parameter is supported as a range table, then it should have minimum and maximum values.

This function should be used to verify that a parameter is supported as a range table before trying to retrieve it as a range table.

Note: Althrough some parameters that are unsupported as range table still report a range (via QCam_GetParamS32Min() or QCam_GetParamS32Max()), it is not recommended that they are used. They are supported for backwards compatibility only.

Header: QCamAPI.h

QCam_IsSparseTable

Determine if a 32-bit parameter is supported as a sparse table.

```
extern QCam_Err QCAMAPI QCam_IsSparseTable
(
    QCam_Settings const  *pSettings,
    QCam_Param           paramKey
);
```

Parameter Descriptions

➔ `pSettings`
A QCam_Settings structure to read (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The 32-bit parameter to check.

function result
qerrSuccess if the parameter is a valid sparse table.

qerrNotSupported if the paramter is not supported as a sparse table.

Discussion

If the parameter is supported as a range table, then it may have missing settable values in between (ie: Binning Modes: [1, 2, x, 4, x, x, x, 8], where x is a missing entry).

This function should be used to verify that a parameter is supported as a sparse table before trying to retrieve it as a sparse table.

Header: QCamAPI.h

QCam_IsSparseTable64

Determine if a 64-bit parameter is supported as a sparse table.

```
extern QCam_Err QCAMAPI QCam_IsSparseTable64
(
    QCam_Settings const  *pSettings,
    QCam_Param64         paramKey
);
```

Parameter Descriptions

➔ `pSettings`
A QCam_Settings structure to read (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The 64-bit parameter to check.

function result
qerrSuccess if the parameter is a valid sparse table.

qerrNotSupported if the paramter is not supported as a sparse table.

Discussion

If the parameter is supported as a range table, then it may have missing settable values in between (ie: Binning Modes: [1, 2, x, 4, x, x, x, 8], where x is a missing entry).

This function should be used to verify that a parameter is supported as a sparse table before trying to retrieve it as a sparse table.

Header: QCamAPI.h

QCam_IsSparseTableS32

Determine if a signed 32-bit parameter is supported as a sparse table.

```
extern QCam_Err QCAMAPI QCam_IsSparseTableS32
(
    QCam_Settings const  *pSettings,
    QCam_ParamS32        paramKey
);
```

Parameter Descriptions

➔ `pSettings`
A QCam_Settings structure to read (can be the QCam_SettingsEx pointer type-cast).

➔ `paramKey`
The signed 32-bit parameter to check.

function result
qerrSuccess if the parameter is a valid sparse table.

qerrNotSupported if the paramter is not supported as a sparse table.

Discussion

If the parameter is supported as a range table, then it may have missing settable values in between (ie: Binning Modes: [1, 2, x, 4, x, x, x, 8], where x is a missing entry).

This function should be used to verify that a parameter is supported as a sparse table before trying to retrieve it as a sparse table.

Header: QCamAPI.h

QCam_LibVersion

Get the version of the QCam driver.

```
extern QCam_Err QCAMAPI QCam_LibVersion
(
    unsigned short       *verMajor,
    unsigned short       *verMinor,
    unsigned short       *verBuild
);
```

Parameter Descriptions

← `verMajor`
A pointer to the major version number

← `verMinor`
A pointer to the minor version number

← `verBuild`
A pointer to the build version number

function result
A result code.

Discussion

This function may be used before QCam_LoadDriver() is called. The version numbers for the QCam driver are synchronized across all platforms.

Header: QCamAPI.h

QCam_ListCameras

List all the cameras that are accessible to the driver.

```
extern QCam_Err QCAMAPI QCam_ListCameras
(
    QCam_CamListItem     *pList,
    unsigned long        *pNumberInList
);
```

Parameter Descriptions

← `pList`

A pointer to an array of QCam_CamListItem strucures. The structures are filled with information about each available camera.

←→ `pNumberInList`

On input, the number of elements in the QCam_CamListItem array.

On output, the number of cameras detected. If this number is smaller or equal to your array length, then this is the number of valid entries in your array. This number may be larger than your array length.

function result

A result code.

Discussion

This function needs to be called before QCam_OpenCamera() is used. It fills out the QCam_CamListItem structures, which are defined as follows:

```
typedef struct
{
    unsigned long           cameraId;
    QCam_qcCameraType       cameraType;
    unsigned long           uniqueId;
    unsigned long           isOpen;
    unsigned long           _reserved[10];
}
QCam_CamListItem;
```

cameraId	Used to identify the camera when calling QCam_OpenCamera().
cameraType	The camera model.
uniqueId	A 32-bit number that is unique to each camera.

isOpen	Non-zero if the camera has already been opened.
_reserved	Reserved for future use.

Header: QCamAPI.h

QCam_LoadDriver

Load and initialize the QCam driver.

```
extern QCam_Err QCAMAPI QCam_LoadDriver
(
    void
);
```

Parameter Descriptions

None

function result
A result code.

Discussion

This function must be called before using any other QCam functions, with the exception of QCam_Version().

Win32: The QcamDriver DLL will either a) be loaded on startup or b) loaded on the first call to a QCam function, depending on your compile options.

Header: QCamAPI.h

QCam_OpenCamera

Open a connection to a camera.

```
extern QCam_Err QCAMAPI QCam_OpenCamera
(
    unsigned long        cameraId,
    QCam_Handle          *pHandle
);
```

Parameter Descriptions

➔ `cameraId`
: The camera id to open. This value is acquired through a call to QCam_ListCameras().

➔ `pHandle`
: A pointer to the camera handle. The handle is only valid if *qerrSuccess* is returned. This handle is valid only for the current process.

function result
: A result code.

Discussion

Once a camera has been opened, it may not be opened a second time by any process. If multiple threads need access to a QCam_Handle, they may share a single instance.

Header: QCamAPI.h

QCam_PreflightSettings

Validate and update a camera state.

```
extern QCam_Err QCAMAPI QCam_PreflightSettings
(
    QCam_Handle      handle,
    QCam_Settings    *pSettings
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔➔ `pSettings`
On input, the camera state.

On output, the camera state with any invalid values corrected.

function result
A result code.

Discussion

The camera state information is contained in the opaque QCam_Settings structure.

This function alters (if need be) your QCam_Settings or QCam_SettingsEx (with type-cast pointer) structure just as QCam_SendSettingsToCam() would do, but without actually changing the camera state.

Header: QCamAPI.h

QCam_QueueFrame

Queue one frame.

```
extern QCam_Err QCAMAPI QCam_QueueFrame
(
    QCam_Handle          handle,
    QCam_Frame           *pFrame,
    QCam_AsyncCallback   callback,
    unsigned long        cbFlags,
    void                 *userPtr,
    unsigned long        userData
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

↔ `pFrame`
On input, a *QCam_Frame* structure, where the *pBuffer* and *bufferSize* fields have been set to a buffer allocated by the caller.

On output, the image is written into the *pBuffer* field of the *QCam_Frame*.

➔ `callback`
A callback function pointer. It may be NULL.

➔ `cbFlags`
Flags to indicate which callbacks the user would like. Logical OR values QCam_qcCallbackFlags. There are two valid flags:

qcCallbackExpose	Provides notification when an exposure is complete (the readout has started). If there was an error during the expose, this callback may not be called.
qcCallbackDone	Provides notification when the frame is complete (the readout has finished).

➔ `userPtr`
A user defined pointer that is passed into the callback.

➔ `userData`
A user defined value that is passed into the callback.

function result
A result code.

Discussion

If the user has provided a callback, it will be called when the image exposure is complete, or when the frame is complete (depending on the flags passed in).

Many frames may be queued at a time. The queue used is shared with QCam_QueueSettings() so any frame captures and/or camera settings changes will be executed in the order they appear on the queue.

If you have set the *qprmDoPostProcessing* parameter then QCam_QueueFrame() will return a QCam_Frame that has been colour balanced, binned and Bayer interpolated.

The following is a list of common result codes returned by this function and a description of why each happened:

qerrInvalidHandle	The camera handle used is not valid.
qerrBufferTooSmall	The frame buffer used is too small to hold the entire image.
qerrLostSync	Something on the FireWire bus has resulted in the frame being out of sync. This frame cannot be used. Isochronous data transmission over FireWire is not guaranteed.
qerrBlackFill	Something on the FireWire bus has resulted in some missing image data. The missing data is filled in with black. Isochronous data transmission over FireWire is not guaranteed.
qerrNotStreaming	Streaming must be on to queue frames. See QCam_SetStreaming for more info.
qerrQueueFull	The queue is full. A maximum of 100 items

may be on the queue at one time.

The following is a list of QCam functions that are safe to call from your callback:

- QCam_QueueSettinsg
- QCam_QueueFrame
- QCam_GetInfo
- QCam_GetParam
- QCam_SetParam
- QCam_GetParamMin
- QCam_GetParamMax
- QCam_PreflightSettings
- QCam_ReadDefaultSettings
- QCam_ReadSettingsFromCam

Calling any other function may result in a deadlock or corruption within QCam.

Win32: Callbacks occur from another thread, and they are always serialized.

For now the flag value in the callback is always zero. This may change in the future.

Header: QCamAPI.h

QCam_QueueSettings

Queue a change to the camera settings.

```
extern QCam_Err QCAMAPI QCam_QueueSettings
(
    QCam_Handle           handle,
    QCam_Settings         *pSettings,
    QCam_AsyncCallback    callback,
    unsigned long         cbFlags,
    void                  *userPtr,
    unsigned long         userData
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔➔ `pSettings`
On input, a *QCam_Settings (*can be a QCam_SettingsEx pointer type-cast) structure that defines the camera state to send to the camera.

On output, the same *QCam_Settings (can be a QCam_SettingsEx pointer type-cast)* structure with any invalid settings corrected. This is done before any callbacks are made.

➔ `callback`
A callback function pointer. It may be NULL.

➔ `cbFlags`
Flags to indicate which callbacks the user would like. There is one valid flag:

qcCallbackDone	Provides notification when the settings change is complete.

➔ `userPtr`
A user defined pointer that is passed into the callback.

➔ `userData`
A user defined value that is passed into the callback.

function result
A result code.

Discussion

If the user has provided a callback, it will be called when the camera state change is complete.

Many frames may be queued at a time. The queue used is shared with

QCam_QueueFrame() so any frame captures and/or camera settings changes will be executed in the order they apprear on the queue.

The following is a list of common result codes returned by this function and a description of why each happened:

qerrInvalidHandle	The camera handle used is not valid.
qerrBadSettings	The settings structure passed in is not valid. Example: format is incorrect, ROI is too small, the trigger mode is different from the current state, or the size field has not been set.
qerrNotStreaming	Streaming must be on to queue frames. See QCam_SetStreaming for more info.
qerrQueueFull	The queue is full. A maximum of 100 items may be on the queue at one time.

The current format, ROI parameters or trigger mode cannot be changed through this function. Use QCam_SetParam() instead.

The following is a list of QCam functions that are safe to call from your callback:

- QCam_QueueSettinsg
- QCam_QueueFrame
- QCam_GetInfo
- QCam_GetParam
- QCam_GetParamMin
- QCam_GetParamMax
- QCam_SetParam
- QCam_GetParamS32
- QCam_GetParamS32Min
- QCam_GetParamS32Max
- QCam_GetParam64
- QCam_GetParam64Min
- QCam_GetParam64Max

- QCam_PreflightSettings
- QCam_ReadDefaultSettings
- QCam_ReadSettingsFromCam
- QCam_InitializeCameraSettingsStruct

Calling any other function may result in a deadlock or corruption within QCam.

Win32: Callbacks occur from another thread, and they are always serialized.

For now the flag value in the callback is always zero. This may change in the future.

Header: QCamAPI.h

QCam_ReadDefaultSettings

Read the default state of the camera.

```
extern QCam_Err QCAMAPI QCam_ReadDefaultSettings
(
    QCam_Handle      handle,
    QCam_Settings    *pSettings
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `pSettings`
A pointer to a *QCam_Settings (*or a type-cast pointer to a QCam_SettingsEx) structure that will contain the default state.

function result
A result code.

Discussion

The default state is specific to each individual camera. Some parameter defaults are factory calibrated.

Header: QCamAPI.h

QCam_ReadSettingsFromCam

Read the current state of the camera.

```
extern QCam_Err QCAMAPI QCam_ReadSettingsFromCam
(
    QCam_Handle      handle,
    QCam_Settings    *pSettings
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `pSettings`
A pointer to a *QCam_Settings (*or a type-cast pointer to a QCam_SettingsEx) structure that will contain the current camera state.

function result
A result code.

Header: QCamAPI.h

QCam_CreateCameraSettingsStruct

Create a camera settings structure to hold the current camera state. Memory for the structure must be allocated statically or dynamically by the user prior to calling this routine. The routine will in turn create and assign resources needed by this structure.

```
extern QCam_Err QCAMAPI QCam_CreateCameraSettingsStruct
(
    QCam_SettingsEx  *pSettings
);
```

Parameter Descriptions

← `pSettings`
A pointer to a *QCam_SettingsEx* structure that will contain the current camera state.

function result
A result code.

Header: QCamAPI.h

QCam_InitializeCameraSettings

Initialize the camera settings structure with the camera default settings. The structure must have been be previously created with QCam_CreateCameraSettingsStruct.

```
extern QCam_Err QCAMAPI QCam_InitializeCameraSettings
(
    QCam_Handle      handle,
    QCam_SettingsEx  *pSettings
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `pSettings`
A pointer to a *QCam_SettingsEx* structure that will contain the current camera state.

function result
A result code.

Header: QCamAPI.h

QCam_ReleaseCameraSettingsStruct

Closes and releases resources used by QCam_SettingsEx Struct. If the user allocated space for the QCam_SettingsStructEx dynamically, then that space must be released by the user after calling this routine.

```
extern QCam_Err QCAMAPI QCam_ReleaseCameraSettingsStruct
(
    QCam_SettingsEx  *pSettings
);
```

Parameter Descriptions

← `pSettings`
A pointer to a *QCam_SettingsEx* structure that contains the current camera state.

function result
A result code.

Header: QCamAPI.h

QCam_ReleaseDriver

Closes and releases any resources used by the QCam driver.

```
extern void QCAMAPI QCam_ReleaseDriver
(
    void
);
```

function result

None

Discussion

Win32: This function does not call FreeLibrary() on the QCamDriver DLL.

After calling QCam_ReleaseDriver(), a call to QCam_LoadDriver() must be made before using any of the QCam functions.

Note: Once this function is called, any cameras that were found with QCam_ListCameras() become invalid.

Header: QCamAPI.h

QCam_SendSettingsToCam

Set the camera state.

```
extern QCam_Err QCAMAPI QCam_SendSettingsToCam
(
    QCam_Handle      handle,
    QCam_Settings    *pSettings
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔➔ `pSettings`
On input, a pointer to a *QCam_Settings* (can be a QCam_SettingsEx pointer type-cast) structure that will contain the current camera state.

On output, any settings values that are invalid will be corrected.

function result
A result code.

Discussion

The state information is contained in the opaque QCam_Settings structure. The values are validated, and modified if necessary, and then sent to the camera.

To check if any values have been modified before they are sent to the camera, the function QCam_PreflightSettings() can be used.

The frame and settings queue must be clear before this function is used.

Header: QCamAPI.h

QCam_SetParam

Set a camera parameter.

```
extern QCam_Err QCAMAPI QCam_SetParam
(
    QCam_Settings    *pSettings,
    QCam_Param       paramKey,
    unsigned long    value
);
```

Parameter Descriptions

➔ `pSettings`
A pointer to a *QCam_Settings* (can be a QCam_SettingEx pointer type-cast) structure that will be modified.

➔ `paramKey`
A key to to select with parameter to modify.

➔ `value`
The new parameter value. It is always an unsigned 32-bit value.

function result
A result code.

Discussion

The opaque QCam_Settings structure represents a camera state.

The camera state will not be changed until the function QCam_SendSettingsToCam() is called.

Header: QCamAPI.h

QCam_SetParam64

Set a 64-bit camera parameter.

```
extern QCam_Err QCAMAPI QCam_SetParam64
(
    QCam_Settings    *pSettings,
    QCam_Param64     paramKey,
    UNSIGNED64       value
);
```

Parameter Descriptions

➔ `pSettings`
A pointer to a *QCam_Settings* (can be a QCam_SettingEx pointer type-cast) structure that will be modified.

➔ `paramKey`
A key to to select with parameter to modify.

➔ `value`
The new parameter value. It is always an unsigned 64-bit value.

function result
A result code.

Discussion

The opaque QCam_Settings structure represents a camera state.

The camera state will not be changed until the function QCam_SendSettingsToCam() is called.

Header: QCamAPI.h

QCam_SetParamS32

Set a signed 32-bit camera parameter.

```
extern QCam_Err QCAMAPI QCam_SetParamS32
(
    QCam_Settings    *pSettings,
    QCam_ParamS32    paramKey,
    signed long      value
);
```

Parameter Descriptions

➔ `pSettings`
A pointer to a *QCam_Settings* (can be a QCam_SettingEx pointer type-cast) structure that will be modified.

➔ `paramKey`
A key to to select with parameter to modify.

➔ `value`
The new parameter value. It is always an signed 32-bit value.

function result
A result code.

Discussion

The opaque QCam_Settings structure represents a camera state.

The camera state will not be changed until the function QCam_SendSettingsToCam() is called.

Header: QCamAPI.h

QCam_SetStreaming

Turn streaming on / off.

```
extern QCam_Err QCAMAPI QCam_SetStreaming
(
    QCam_Handle      handle,
    unsigned long    enable
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `enable`
Non-zero to enable streaming, zero to disable streaming.

function result
A result code.

Discussion

Calling this function will tell the camera to either start or stop streaming data. To receive an image from this stream, either QCam_GrabFram() or QCam_QueueFrame() must be called.

When using QCam_QueueFrame or QCam_QueueSettings(), streaming should be turned on. If it is not, grabbing frames will be slower.

When a camera is streaming, the frame rate is higher, but there is an overhead to process the incoming FireWire packets even if frames are not being grabbed.

If streaming is off, the driver will automatically start and stop streaming for each frame grab.

The frame and settings queue must be clear before using this function.

Header: QCamAPI.h

QCam_TranslateSettings

Translate one settings structure so that another camera may use it.

```
extern QCam_Err QCAMAPI QCam_TranslateSettings
(
    QCam_Handle      handle
    QCam_Settings    *pSettings
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `pSettings`
A pointer to a camera settings structure to translate and set the new camera to.

function result
A result code.

Header: QCamAPI.h

QCam_Trigger

Software trigger to start a new exposure.

```
extern QCam_Err QCAMAPI QCam_Trigger
(
    QCam_Handle      handle
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera..

function result
A result code.

Discussion

Before callling this function, frames must be queued up with QCam_QueueFrame() to ensure that the image is not missed.

QCam_GrabFrame() may not be used in conjunction with QCam_Trigger().

The camera must not be in freerun mode (*qcTriggerFreerun)*.

Header: QCamAPI.h

Image API Functions

QCam_is16bit

Is the image format 16-bit?

```
bool QCAMAPI QCam_is16bit
(
    unsigned long    format
);
```

Parameter Descriptions

➔ `format`
The image format to test.

function result
True if the format is 16-bit.

False if the format is not 16-bit.

Header: QCamImgfnc.h

QCam_is3Color

Is the image format 3-colour? (ie: rgb-filter)

```
bool QCAMAPI QCam_is3Color
(
    unsigned long    format
);
```

Parameter Descriptions

➔ `format`
The image format to test.

function result
True if the format is 3-colour.

False if the format is not 3-colour.

Header: QCamImgfnc.h

QCam_isBayer

Is the image format Bayer?

```
bool QCAMAPI QCam_isBayer
(
    unsigned long    format
);
```

Parameter Descriptions

➔ `format`
The image format to test.

function result
True if the format is Bayer.

False if the format is not Bayer.

Header: QCamImgfnc.h

QCam_isColor

Is the image format colour? (ie: Bayer or 3-colour)

```
bool QCAMAPI QCam_isColor
(
    unsigned long    format
);
```

Parameter Descriptions

➔ `format`
The image format to test.

function result
True if the format is colour.

False if the format is not colour.

Header: QCamImgfnc.h

QCam_isMonochrome

Is the image format monochrome?

```
bool QCAMAPI QCam_isMonochrome
(
    unsigned long    format
);
```

Parameter Descriptions

➔ `format`
The image format to test.

function result
True if the format is monochrome.

False if the format is not monochrome.

Header: QCamImgfnc.h

QCam_CalcImageSize

Calculate the image size, in bytes.

```
unsigned long QCAMAPI QCam_CalcImageSize
(
    unsigned long    format,
    unsigned long    width,
    unsigned long    height
);
```

Parameter Descriptions

➔ `format`
The format of the image.

➔ `width`
The width of the image in pixels.

➔ `height`
The height of the image in pixels.

function result
The image size in bytes.

Header: QCamImgfnc.h

QCam_BayerToRgb

Interpolate the Bayer image to some type of RGB format.

```
void QCAMAPI QCam_BayerToRgb
(
    QCam_qcBayerInterp       algorithm,
    QCam_Frame               *pFrameIn,
    QCam_Frame               *pFrameOut
);
```

Parameter Descriptions

➔ `algorithm`
The interpolation method to use

➔ `pFrameIn`
A pointer to the frame to process.

➔➔ `pFrameOut`
On input, a frame structure with the *pBuffer, bufferSize and format* fields set.

On output, a pointer to the converted frame. The image is placed into the *pBuffer* field.

function result
None.

Discussion

The *format* field of *pFrameOut* determines the desired colour output format. It must be the same bit depth as the intput format.

The following is a list of the different interpolation algorithms available, along with a description of each:

qcBayerInterpNone	Does not perform any interpolation.
qcBayerInterpAvg4	Averages the 4 neighboring greens and reds at every blue pixel. Averages the 2 neighboring blues and reds at every green pixel. Averages the 4 neighboring greens and blues at every red pixel.
qcBayerInterpFast	Nearest neighbor algorithm. Averages the 2

	neighboring red and 2 neighboring blue pixel at every green pixel. Averages the 2 neighboring vertical or horizontal green pixel at every red/blue pixel depending on the difference between the next 2 horizontal or vertical red/blue pixels.
qcBayerBiCubic	BiCubic interpolation generates each target pixel by interpolating from the nearest sixteen mapped source pixels. Two cubic interpolation polynomials, one for each plane direction, are used. This algorithm yields a good balance between accuracy in detail preservation and smoothness for resampled images. Its performance is quite is very demanding on processor speed and memory usage.
qcBayerBiCubic_Faster	This uses the same algorithm as the regular BiCubic except that this version is optimized. The optimization involves faster memory copying as well as converting most floating point operations to use integers. It is significantly faster than the regular version.

Header: QCamImgfnc.h

QCam_BayerZoomVert

Expand a Bayer pattern in the vertical direction by a multiple of 2.

```
QCam_Err QCAMAPI QCam_BayerZoomVert
(
    unsigned char        factor,
    QCam_Frame           *pFrameIn,
    QCam_Frame           *pFrameOut
);
```

Parameter Descriptions

➔ `factor`
The multiple to expand the image by. Currently only a factor of 2 is supported.

➔ `pFrameIn`
A pointer to the frame to process.

➔➔ `pFrameOut`
On input, a frame structure with the *pBuffer, bufferSize and format* fields set.

On output, a pointer to the converted frame. The image is placed into the *pBuffer* field.

function result
None.

Discussion

The *pBuffer* field in *pFrameOut* must be *factor* times the original image size.

Example: If factor is 2, then the buffer must be twice as big.

The *format* field must be the same as the input format. The formats allowed are *qfmtBayer8* and *qfmtBayer16*

Header: QCamImgfnc.h

QCam_AutoExpose

Run an automatic auto expose algorithm.

```
extern QCam_Err QCAMAPI QCam_AutoExpose
(
      QCam_Settings              *pSettings,
      unsigned long              xOrig,
      unsigned long              yOrig,
      unsigned long              width,
      unsigned long              height
);
```

Parameter Descriptions

➔ `pSettings`
A pointer to a camera settings structure.

➔ `xOrig`
X coordinate of the top left corner of the ROI to be used for the auto expose calculations.

➔ `xOrig`
Y coordinate of the top left corner of the ROI to be used for the auto expose calculations.

➔ `width`
The width of the ROI to be used for the auto expose calculations.

➔ `height`
The height of the ROI to be used for the auto expose calculations.

function result
A result code.

Discussion

Before calling this function, make sure that streaming is turned off on the camera and there are no settings changes queued up. Also make sure there are not threads in parallel calling QCam_GrabFrame().

If you pass only zeros as the ROI parameters (`xOrig=0, yOrig=0, width=0, height=0`) then the Auto Expose algorithm will be applied to the full image.

Header: QCamAPI.h

QCam_WhiteBalance

Run an automatic white balance algorithm

```
extern QCam_Err QCAMAPI QCam_WhiteBalance
(
      QCam_Settings              *pSettings,
      unsigned long              xOrig,
      unsigned long              yOrig,
      unsigned long              width,
      unsigned long              height
);
```

Parameter Descriptions

➔ `pSettings`
A pointer to a camera settings structure.

➔ `xOrig`
X coordinate of the top left corner of the ROI to be used for the white balance calculations.

➔ `xOrig`
Y coordinate of the top left corner of the ROI to be used for the white balance calculations.

➔ `width`
The width of the ROI to be used for the white balance calculations.

➔ `height`
The height of the ROI to be used for the white balance calculations.

function result
A result code.

Discussion

Before calling this function, make sure that streaming is turned off on the camera and there are no settings changes queued up. Also make sure there are not threads in parallel calling QCam_GrabFrame().

In order for QCam_WhiteBalance() to succeed you must specify an ROI or the call will return an error stating ROI is too small.

If you are using a Bayer Color Camera then in order to white balance the images properly you will need to set the *qprmDoPostProcessing* to 1 in the settings structure. This flag will enable automatic Bayer interpolation, horizontal binning (for Micropublishers) and white balancing on each new image.

The *qprmPostProcessBayerAlgorithm* parameter will allow you to specify which interpolation algorithm will be used to do the Bayer Interpolation.

The *qprmPostProcessImageFormat* specifies which image format the interpolation will produce.

If you wish to do your own color balancing but make use of the automatic post processing you may set the *qprmPostProcessGainRed*, *qprmPostProcessGainGreen*, *qprmPostProcessGainBlue* parameters in the settings structure individually.

Note: ROI must be at least four pixels wide and four pixels high.

Header: QCamAPI.h

QCam_PostProcessSingleFrame

Post process a single frame.

```
extern QCam_Err QCAMAPI QCam_PostProcessSingleFrame
(
      QCam_Handle                handle,
      QCam_Settings              settings,
      QCam_Frame                 inFrame,
      QCam_Frame                 outFrame,
);
```

Parameter Descriptions

➔ `handle`
A handle to an open camera.

➔ `settings`
A pointer to a camera settings structure.

➔ `inFrame`
The original un-processed frame.

➔ `outFrame`
The post processed frame. The pBuffer is allocated by QCam.

function result
A result code.

Discussion

Before calling this function, make sure that qprmPostProcessImageFormat and qprmPostProcessBayerAlgorithm are set.

This function will allow the user to see the raw image data that was returned from the camera before allowing QCam to post process the image.

Note: This function does not support the Go 21 in oversample mode.

Header: QCamAPI.h