ccAcqFifo

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

class ccAcqFifo : public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Cognex-supplied classes only
Archiveable	No

You create **ccAcqFifo** objects by calling **ccStdVideoFormat::newAcqFifoEx()** which returns various instantiations derived from this class. You use **ccAcqFifo** methods to acquire images for all concrete fifo types.

Notes

Using **ccAcqFifo::complete()** in derived classes to handle acquired images is now replaced by **ccAcqFifo::completeAcq()**. The older method is retained for backward compatibility, but you should use **ccAcqFifo::completeAcq()** for all new code.

Constructors/Destructors

ccAcqFifo

```
virtual ~ccAcqFifo();

ccAcqFifo(ccAcqFifo&);
```

•
```
virtual ~ccAcqFifo();
```

Destroys the FIFO and cancels all outstanding acquisitions on it.

•
```
ccAcqFifo(ccAcqFifo&);
```

Copy constructor.

Enumerations

ceStartReqStatus

```
enum ceStartReqStatus;
```

These constants provide additional information about the state of an acquisition request after a call to **baseComplete()** or **complete()** in the derived classes.

Value	Meaning
ckStartReqWasServiced	• A new call to **start()** is needed to get the next image. • The returned *appTag* is valid. • The **complete()** call is in sync with the other FIFOs of the master-slave sequence.
ckStartReqNotServiced	• A new call to **start()** is not needed to get the next image. • The returned *appTag* is not valid. • The **complete()** call is out of sync with the other FIFOs of the master-slave sequence.

enum

```
enum {kMaxOutstanding = 32};
```

The maximum number of outstanding acquisitions supported per FIFO. Attempting to have more than this number of acquisitions outstanding at one time causes either **start()** to return false (if the trigger model specifies *ckRequestAcquireOrFail* as its **startAction()**) or a *ccAcqFailure::isMissed* failure otherwise.

Public Member Functions

start

```
bool start(c_UInt32 appTag = 0);
```

Requests that an acquisition be started as soon as possible. The request is added to the FIFO of outstanding acquisitions. A copy of the current property values is stored with the request, so that subsequent changes to this FIFO's properties have no effect on the outstanding acquisitions.

Returns true to indicate the start request was successful. A return of false means the resources were not available and the requested acquisition will not take place.

appTag is an arbitrary value supplied by the application for identifying this acquisition request. The value is returned by the matching **complete()** or **baseComplete()** invocation. The value is not interpreted by **ccAcqFifo**. Typically, the *appTag* is used to facilitate FIFO order integrity verification.

Parameters

appTag	An application defined value that identifies this acquisition request.

Notes

The exact behavior of **start()** is controlled by **ccTriggerModel::startAction()**. See **ceStartAction** on page 2568.

See **prepare()** for a description of what it means to start an acquisition "as soon as possible."

triggerEnabled() must be true for acquisitions to proceed.

Throws

ccAcqFifo::StartNotAllowed
The selected trigger model does not allow **start()** to be invoked. See **ceStartAction** on page 2568.

■ ccAcqFifo

baseComplete

```
cc_PelBuffer* baseComplete (
  const ccAcqFifo::CompleteArgs& args);

cc_PelBuffer* baseComplete (
  ccAcqFailure* failure=0,
  c_UInt32* appTag = 0,
  bool makeLocal = true,
  double maxWait = HUGE_VAL,
  bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Notes

Using **baseComplete()** or **complete()** in derived classes to handle acquired images is now replaced by **ccAcqFifo::completeAcq()**. The older method is retained for backward compatibility, but you should use **ccAcqFifo::completeAcq()** for all new code.

Classes derived from this class override this function to return a more specific **ccPelBuffer<P>** type. Derived classes typically also supply a function, **complete()**, which is similar to **baseComplete()** but which returns a **ccPelBuffer** object rather than a pointer to one.

The returned pel buffer is a window onto the valid pixels of a pelroot. The underlying pelroot may be larger than this window. For example, the width of the underlying pelroot is often larger to accommodate DMA transfer alignment restrictions imposed by each frame grabber. In this case, the returned pel buffer is a window onto the valid pixels, and the underlying pelroot is larger and contains invalid pixel data. Avoid resizing or repositioning the pel buffer to include the invalid pixel data, since there is no guarantee on the contents of this memory.

The first **baseComplete()** overload takes a **ccAcqFifo::CompleteArgs** object to specify its parameters. Using **CompleteArgs** allows you to specify any parameter in any order without having to specify the preceding parameters. For more information on the **baseComplete()** parameters see **CompleteArgs** on page 3249.

Using the second **baseComplete()** overload you specify individual parameters as call arguments. This syntax has the following limitations, compared to the first overload:

1. Arguments must be specified in the given order, even when not making use of a particular argument. For example, if you wish only to change the *maxWait* timeout value, you must also specify the three preceding parameters.
2. You cannot specify the use of a **ccAcquireInfo** object to contain the results of the image acquisition. The **ccAcquireInfo::triggerNum()** feature does not have an analogous argument in the old form, and thus cannot be used with the old form.

The recommended way to examine or test the results of an image acquisition is to specify a **ccAcquireInfo** object to contain the results. This object is specified as a **CompleteArgs** parameter, as shown in the second example on page 270. For more information on **ccAcquireInfo** objects, see **ccAcquireInfo** on page 301.

•
```
cc_PelBuffer* baseComplete (
  const ccAcqFifo::CompleteArgs& args);
```

Specify parameters in a **CompleteArgs** object. See **CompleteArgs** on page 3249.

Returns the image of the oldest completed outstanding image acquisition in the FIFO, and removes it from the FIFO. If there are no completed acquisitions in the FIFO, this function waits for one.

Returns null if the acquisition failed or if the **ccAcqFifo::CompleteArgs().maxWait()** period has elapsed. In this case, the **ccAcquireInfo** object you specified with **ccAcqFifo::CompleteArgs** contains the reason the acquisition failed.

Parameters

args	A completion arguments object.

Throws

ccPel::BadWindow

You specified a region of interest, but the intersection of the ROI and the entire window was a null rectangle. See **ccRoiProp** on page 2259.

Example

This code fragment gets an image from an outstanding acquisition:

```
auto_ptr<cc_PelBuffer> pb(fifo->baseComplete());
if (pb.get())
  processImage(*pb);
```

This code fragment determines how the acquisition failed:

```
ccAcquireInfo info;
ccPtrHandle<cc_PelBuffer> pb = fifo->baseComplete
  (ccAcqFifo::CompleteArgs().acquireInfo(&info));
if (pb)
  processImage(*pb);
else if (info.failure().isTimeout())
  // Clean up after the timeout
else
  // Report or log an unexpected acquisition failure
```

•
```
cc_PelBuffer* baseComplete (
  ccAcqFailure* failure=0,
  c_UInt32* appTag = 0,
  bool makeLocal = true,
  double maxWait = HUGE_VAL,
  bool autoStart = false,
  ceStartReqStatus* startReqStatus = 0);
```

Specify the parameters as arguments.

Parameters

failure	See the description of **ccAcquireInfo::failure()** on page 302.
appTag	See the description of **ccAcquireInfo::appTag()** on page 302.
makeLocal	See the description of **ccAcqFifo::CompleteArgs().makeLocal()** on page 3251.
maxWait	See the description of **ccAcqFifo::CompleteArgs().maxWait()** on page 3252.
autoStart	Deprecated parameter. See the description of **ccAcqFifo::CompleteArgs().autoStart()** on page 3253.
startReqStatus	See the description of **ccAcqFifo::CompleteArgs().startReqStatus()** on page 3252.

completeAcq

```
virtual ccAcqImagePtrh completeAcq(
  const CompleteArgs& args = CompleteArgs()) = 0;
```

Gets the image of the oldest outstanding acquisition. This call blocks if there is no outstanding completed acquisition.

The function returns the acquired image. If the acquisition failed or the *maxWait* period elapsed the image will be unbound.

Parameters

args	A completion arguments object.

isValid

```
bool isValid(ccAcqProblem* problem = 0) const;
```

Returns true if the FIFO is configured properly and is able to perform an image acquisition.

Parameters

problem	If this function returns false, this optional argument contains the reason that the FIFO was not configured properly.

Notes

isValid() performs a more comprehensive check of a FIFO's master/slave configuration than the **ccTriggerProp::triggerMaster()** and **ccTriggerProp::couldSlaveTo()** methods. If a master/slave configuration does not perform as expected, use **isValid()** to check whether the FIFOs are configured properly. If **isValid()** returns false, use **ccAcqProblem::hasInvalidMaster()** or **ccAcqProblem::hasInvalidSlave()** to diagnose the problem.

isIdle

```
bool isIdle() const;
```

Returns true if there are no outstanding acquisitions on the FIFO.

isWaiting

```
bool isWaiting() const;
```

Becomes true when the FIFO begins waiting for **start()**, and returns to false when hardware resources are allocated and all setup delays are finished.

isAcquiring

```
bool isAcquiring() const;
```

Returns true if the oldest outstanding acquisition is waiting for a trigger signal or is acquiring an image. For master/slave acquisitions, it becomes true only after the master and all slaves are ready for acquisition.

isMovable

```
bool isMovable() const;
```

Returns true if the oldest outstanding acquisition has progressed to the point where the camera's field of view can be changed without affecting the acquired image. For example, for strobed and shuttered acquisitions, this means that the strobe or the shutter has been fired.

When an acquisition enters this state, the acquisition software invokes the move-part callback function that you can specify with the move-part callback property. See **ccMovePartCallbackProp** on page 1791.

isComplete

```
bool isComplete() const;
```

Returns true if the oldest outstanding acquisition has completed, perhaps unsuccessfully. The next invocation of **baseComplete()** returns the image if the acquisition was successful or a null pointer if it was not.

When an acquisition enters this state, the acquisition software invokes the acquisition complete callback function that you can specify with the acquisition complete callback property. See **ccCompleteCallbackProp** on page 1057.

pendingAcqs

```
c_Int32 pendingAcqs() const;
```

Returns the number of acquisitions in the *pending* state. This is the number of acquisitions requested by **start()** for which acquisition has not started. To achieve high frame rate acquisition in manual trigger mode, the FIFO must always have one or more pending acquisitions.

completedAcqs

```
c_Int32 completedAcqs() const;
```

Returns the number of acquisitions in the *completed* state. This is the number of completed acquisitions for which the user has not called **complete()**. This is similar to **isComplete()**, except it indicates if more than one completed acquisition is available. If images are being acquired faster than they are being consumed, the number returned by **completedAcqs()** will begin increasing. If left unchecked, this will eventually result in *isMissed* or other errors. **completedAcqs()** can be used to control production line speed to maximize the inspection rate without generating errors.

availableAcqs

```
c_Int32 availableAcqs() const;
```

Returns the number of acquisitions in the *available* state. This method returns an upper bound on the number of additional acquisitions that may be added to the FIFO before errors occur. In manual or semi-automatic trigger mode, calling **start()** decreases the number of available acquisitions. In auto trigger mode, each trigger decreases the number of available acquisitions. In all trigger modes, calling **complete()** and releasing pel buffers increases available acquisitions.

Notes

The number returned by **availableAcqs()** is an upper bound and not a guaranteed minimum. It is intended to be used to detect resource leaks during debugging.

The sum of (pending + completed + available) is not a constant. Acquisitions in progress are neither pending nor completed.

Calling **start()** does not always decrease **availableAcqs()** by one.

Actions of other FIFOs may affect **availableAcqs()**.

pendingAcqs() and **completedAcqs()** function the same on all platforms and CVL revisions. **availableAcqs()** may behave differently between platforms and CVL revisions. For example, the way available image memory affects **availableAcqs()** can vary.

prepare

```
bool prepare(double maxWait = HUGE_VAL);
```

This function prepares the video subsystem as needed for this FIFO, waiting up to *maxWait* seconds before returning. It returns true if successful and false if not successful. If **prepare()** returns false, do not attempt an acquisition with this FIFO, as it will fail and likely hang.

Calling **prepare()** does not eliminate or reduce the setup time, but allows the same setup time to occur prior to calling **start()**. This is useful if an acquisition cannot be started immediately, but you want to get the hardware ready now. If an acquisition can start immediately, there is no performance benefit to calling **prepare()** before **start()**. In this case, the primary use of **prepare()** is to detect hardware setup problems, such as a missing camera.

For an acquisition to take place, the FIFO requires that a certain subset of the video subsystem be in the proper state. For example, a FIFO created to acquire images in RS-170 640x480 format, requires that the video subsystem switch to RS-170 640x480 timing. If the video subsystem section selected by this FIFO is already in this state, the FIFO is considered prepared, and acquisitions can begin immediately.

If the FIFO is not prepared when an acquisition starts, the acquisition software starts preparing the video subsystem automatically and blocks all subsequent acquisitions for that FIFO until the FIFO is prepared or until the timeout limit is reached. Be aware that certain state transitions of the video subsystem take a significant amount of time (up to hundreds of milliseconds) to complete.

The **prepare()** and **isPrepared()** functions let you make sure that an acquisition can start immediately.

Notes

isPrepared() is deprecated..

In a typical application, the acquisition FIFOs are always prepared and all acquisitions start immediately, so most applications do not need to use the **prepare()** function.

Parameters

maxWait	The number of seconds to wait until the video subsystem is prepared before returning. The value *HUGE_VAL* (defined in *<math.h>)* indicates that the system should wait indefinitely. The FIFO should be idle before calling **prepare()**. If it is not idle (starts are queued, or you are in auto trigger mode with triggers enabled), **prepare()** will assume the hardware is already prepared and will return true immediately. When the FIFO is idle there should be no wait, so Cognex recommends you always set *maxWait* to 0.

Notes

On most platforms, *maxWait* has a granularity between 1 millisecond and 1 microsecond.

If **ccTriggerProp::triggerEnable()** is false and **start()** calls are queued, **prepare()** will return immediately without preparing the FIFO. If you use **prepare()** you should call **prepare()** and then **start()**, not the reverse.

Calling **prepare()** on a master or slave FIFO prepares all the associated master and slave FIFOs. You could prepare each master and slave FIFO individually, but that would be redundant.

You do not need to call **prepare()** when using automatic triggering. The acquisition software performs hardware preparation for you when **triggerEnable()** is true.

flush

```
void flush();
```

Discards all outstanding acquisitions, leaving this FIFO in the idle state. If an acquisition is in progress, it is cancelled and discarded. If **ccTriggerProp::triggerEnable()** is true, this function disables triggers before discarding all acquisitions, then re-enables triggers.

properties

```
virtual ccAcqProps& properties()= 0;
```

Returns a reference to the FIFO's properties object.

Notes

The reference that this function returns is not **const** so that you can invoke appropriate **ccAcqProps** member functions to set individual properties. Derived classes typically override this function to return a more specific type derived from **ccAcqProps**.

propertyQuery

```
ccAcqPropertyQuery& propertyQuery();
```

Returns a property query object that you can use to test whether the FIFO supports a particular property. If the property is supported, this function returns a valid property object. The returned object may be used to set property values. Both pointer and reference semantics are supported.

Example

To set the timeout property only if this FIFO supports it, you would write:

```
ccTimeoutProp* timeoutProp = fifo->propertyQuery();
if (timeoutProp)
  timeoutProp->timeout(10.0);
```

Or, using reference semantics, you would write:

```
try
{
  ccTimeoutProp& timeoutProp = fifo->propertyQuery();
  timeoutProp.strobeEnable(true);
}
catch (cmStd bad_cast&)
{
  // The timeout property is not supported.
}
```

triggerModel

```
void triggerModel(const ccTriggerModel& model);

void triggerModel(ccTriggerModelPtrh model);

const ccTriggerModel& triggerModel() const;
```

•

```
void triggerModel(const ccTriggerModel& model);
```

Sets the trigger model for the current FIFO. The default model is **cfManualTrigger()**. See **ccTriggerModel** on page 2565.

Parameters

model	A trigger model object, usually created with one of the global functions **cfManualTrigger()**, **cfAutoTrigger()**, **cfSemiTrigger()**, or **cfSlaveTrigger()**.

- `void triggerModel(ccTriggerModelPtrh model);`

Sets the trigger model for the current FIFO. Cognex recommends that custom trigger models be set using the **ccTriggerModelPtrh** overload. Built-in trigger models (such as **cfManualTrigger**) can only be set using the **ccTriggerModel&** overload.

Parameters

model	The name of your custom trigger model object.

- `const ccTriggerModel& triggerModel() const;`

Returns the currently set trigger model for this FIFO.

See also **cfManualTrigger()** on page 3113, **cfAutoTrigger()** on page 2939, **cfSemiTrigger()** on page 3221, and **cfSlaveTrigger()** on page 3229.

triggerEnable

```
void triggerEnable(bool enable);

bool triggerEnable() const;
```

- `void triggerEnable(bool enable);`

Enables or disables triggers. The default is true.

Invoking **triggerEnable()** for a master camera channel or any of its slaves has the effect of invoking **triggerEnable()** for all. That is, masters and slaves have the same **triggerEnable()** setting.

Parameters

enable	In general, true allows acquisitions to happen and false prevents acquisitions from happening, depending on the current trigger model and its state. For more information on each trigger model, see **cfManualTrigger()** on page 3113, **cfAutoTrigger()** on page 2939, **cfSemiTrigger()** on page 3221, and **cfSlaveTrigger()** on page 3229.

- `bool triggerEnable() const;`

Returns true if triggers are enabled, false otherwise.

movePartInfoCallback

```
void movePartInfoCallback(const ccCallbackAcqInfoPtrh&);

const ccCallbackAcqInfoPtrh& movePartInfoCallback() const;
```

- `void movePartInfoCallback(const ccCallbackAcqInfoPtrh&);`

This function is one of two ways to register a callback class for the movable state:

Method	Features	See
ccAcqFifo:: movePartInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	this section
ccMovePartCallbackProp	Calls your callback function.	*ccMovePartCallbackProp* on page 1791

This function effectively registers a callback function that you want the acquisition engine to call when the acquisition enters the movable state (that is, when **ccAcqFifo::isMovable()** returns true). The movable state is when the camera's field of view can be changed without affecting the acquired image.

An unbound handle means no callback will occur.

This function actually registers, not a function, but a callback class you define. Your callback class contains your callback function, defined as an override of **operator()()** of that class.

A **ccCallbackAcqInfo** class is a special case of a **ccCallback1** class in which an instance of **ccAcquireInfo** is passed as an argument. This allows your callback function to use the information about the current state of the acquisition stored in the **ccAcquireInfo** object.

The callback function you write should set flags or semaphores in your application that allow your program to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

In general, an acquisition enters the movable state during the acquisition's next-to-last vertical blank interval. With an RS-170 camera, for example, an acquisition would enter the movable state 17 ms (one field time) before it enters the complete state.

Not all frame grabbers can detect the next-to-last vertical blank interval. When using one of these frame grabbers, the acquisition enters the movable and complete states at the same time. In such cases, the move-part callback is always invoked first, followed immediately by the completion callback.

On the MVS-8100M, MVS-8100C, and MVS-8100C/CPCI (but not the MVS-8100M+ when using CCF-based video formats), you can select between default and optimum timing of the move-part callback function's execution. See **cc8100m::movePartTimingChoice** on page 194.

For a further discussion, see *Using Callback Functions* in the *Acquiring Images* chapter of the *CVL User's Guide.*

Parameters

ccCallbackAcqInfoPtrh
Pointer handle referencing an instance of a callback class you've defined, that contains the callback function you have defined as an override of **operator()()** for that class.

•
```
const ccCallbackAcqInfoPtrh& movePartInfoCallback() const;
```

Returns a pointer handle to the registered callback class instance for this acquisition.

completeInfoCallback

```
void completeInfoCallback(const ccCallbackAcqInfoPtrh&);

const ccCallbackAcqInfoPtrh& completeInfoCallback() const;
```

- `void completeInfoCallback(const ccCallbackAcqInfoPtrh&);`

This function is one of two ways to register a callback class for the complete state:

Method	Features	See
ccAcqFifo:: completeInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	this section
ccCompleteCallbackProp	Calls your callback function.	*ccCompleteCallbackProp* on page 1057

This function effectively registers a callback function that you want the acquisition engine to call when the image acquisition is complete. An acquisition enters the complete state (**ccAcqFifo::isComplete()** returns true) when the next invocation of **ccAcqFifo::baseComplete()** would return the acquired image (if the acquisition was successful) or null (if the acquisition was not successful).

An unbound handle means no callback will occur.

This function actually registers, not a function, but a callback class you define. Your callback class contains your callback function, defined as an override of **operator()()** of that class.

A **ccCallbackAcqInfo** class is a special case of a **ccCallback1** class in which an instance of **ccAcquireInfo** is passed as an argument. This allows your callback function to use the information about the current state of the acquisition stored in the **ccAcquireInfo** object.

The callback function you write should set flags or semaphores in your application that allow your program to proceed to another state the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

For a further discussion, see *Using Callback Functions* in the *Acquiring Images* chapter of the *CVL User's Guide.*

Parameters

ccCallbackAcqInfoPtrh
: Pointer handle referencing an instance of a callback class you've defined, that contains the callback function you have defined as an override of **operator()()** for that class.

-
  ```
  const ccCallbackAcqInfoPtrh& completeInfoCallback() const;
  ```

Returns a pointer handle to the registered callback class instance for this acquisition.

overrunInfoCallback

```
void overrunInfoCallback(const ccCallbackAcqInfoPtrh&);

const ccCallbackAcqInfoPtrh& overrunInfoCallback() const;
```

-
  ```
  void overrunInfoCallback(const ccCallbackAcqInfoPtrh&);
  ```

  This function is one of two ways to register a callback class for the overrun state:

Method	Features	See
ccAcqFifo:: overrunInfoCallback()	Calls your callback function, passing a **ccAcquireInfo** class.	this section
ccOverrunCallbackProp	Calls your callback function.	*ccOverrunCallbackProp* on page 1891

This function effectively registers a callback function that you want the acquisition engine to call if an overrun occurs. Generally, overrun errors and **isMissed()** errors occur when triggers occur too fast. See **ccAcqFailure::isOverrun()** and **ccAcqFailure::isMissed()** for more information about the overrun state.

An unbound handle means no callback will occur.

This function actually registers, not a function, but a callback class you define. Your callback class contains your callback function, defined as an override of **operator()()** of that class.

A **ccCallbackAcqInfo** class is a special case of a **ccCallback1** class in which an instance of **ccAcquireInfo** is passed as an argument. This allows your callback function to use the information about the current state of the acquisition stored in the **ccAcquireInfo** object.

The callback function you write should set flags or semaphores in your application that allow your program to handle the overrun condition the next time it executes. Your callback function is executed internally by the acquisition engine, which cannot proceed until your callback function returns. Callback functions should be short and quick, and *must not block*. Callback functions should not call CVL routines such as **start()** and **complete()**. The time taken to execute your callback function blocks the acquisition engine.

For a further discussion, see *Using Callback Functions* in the *Acquiring Images* chapter of the *CVL User's Guide.*

Parameters

ccCallbackAcqInfoPtrh
: Pointer handle referencing an instance of a callback class you've defined, that contains the callback function you have defined as an override of **operator()()** for that class.

• `const ccCallbackAcqInfoPtrh& overrunInfoCallback() const;`

Returns a pointer handle to the registered callback class instance for this acquisition.

videoFormat `virtual const ccVideoFormat& videoFormat() const = 0;`

Returns the video format for which this FIFO was constructed.

frameGrabber `virtual ccFrameGrabber& frameGrabber() const = 0;`

Returns the frame grabber for which this FIFO was constructed.

hardwareImagePoolSize

```
c_Int32 hardwareImagePoolSize() const;

void hardwareImagePoolSize(c_Int32 numberOfImages);
```

• `c_Int32 hardwareImagePoolSize() const;`

Returns the size of the MVS-8600 hardware image pool in number of images.

• `void hardwareImagePoolSize(c_Int32 numberOfImages);`

Sets the size, in number of images, of the MVS-8600 hardware image pool used the by the acquisition FIFO. A larger image pool size reduces the likelihood of acquisition errors due to system latency or high CPU load. A smaller pool size conserves memory.

For area scan cameras the amount of memory, in bytes, used for the image pool is:

videoFormatWidth * videoFormatHeight * bytesPerPixel * numberOfImages.

For line scan cameras the amount of memory in bytes is:

videoFormatWidth * imageROIHeight * bytesPerPixel * numberOfImages.

In most cases, a minimum pool size of 4 images is recommended. In the case of images larger than 32MB, a pool size of 2 or 3 images may be used to conserve memory if the acquisition rate allows..

Parameters

numberOfImages

The number of images to allocate in the hardware pool. The lowest number allowed is 2 images. Specifying zero sets the number of images to **defaultHardwareImagePoolSize()**.

Notes

This function applies only to the MVS-8600 and MVS-8600e frame grabbers.

If there isn't enough memory available to create an image pool of the given size, **prepare()** returns False, **completeAcq()** fails, and **isAbnormal()** is True with error code 2. In this case, try reducing the pool size. If the pool is exhausted at runtime, **completeAcq()** fails, and **isAbnormal()** is True with error code 7. In this case, try increasing the pool size.

Throws

ccAcqFifo::BadParams

numberOfImages is 1 or less than zero,.

defaultHardwareImagePoolSize

```
c_Int32 defaultHardwareImagePoolSize() const;
```

Returns the default number of images available in the MVS-8600 image pool. For line scan acquisition default the number of images depends on the image height which you can adjust with the region of interest (ROI) setting.

Notes

This function applies only to the MVS-8600 and MVS-8600e frame grabbers.

Deprecated Members

isPrepared

```
bool isPrepared() const;
```

Returns true if the resources that the FIFO needs to perform an acquisition are ready. If **isPrepared()** returns true, an acquisition request begins immediately.

This function is deprecated in CVL 6.2 cr10.

Friends

ccDirectDrawSurfacePool

```
friend class ccDirectDrawSurfacePool;
```

cc_FGDisplay

```
friend class cc_FGDisplay;
```

Typedefs

ccAcqFifoPtrh

```
typedef ccPtrHandle<ccAcqFifo> ccAcqFifoPtrh;
```

ccAcqFifoPtrh_const

```
typedef ccPtrHandle_const<ccAcqFifo> ccAcqFifoPtrh_const;
```