Vimba C++

Manual

V1.4
2015-11-10

Allied Vision Technologies GmbH
Taschenweg 2a
D-07646 Stadtroda / Germany

Allied Vision

Legal Notice

Trademarks

Unless stated otherwise, all trademarks appearing in this document of Allied Vision Technologies are brands protected by law.

Warranty

The information provided by Allied Vision is supplied without any guarantees or warranty whatsoever, be it specific or implicit. Also excluded are all implicit warranties concerning the negotiability, the suitability for specific applications or the non-breaking of laws and patents. Even if we assume that the information supplied to us is accurate, errors and inaccuracy may still occur.

Copyright

All texts, pictures and graphics are protected by copyright and other laws protecting intellectual property. It is not permitted to copy or modify them for trade use or transfer, nor may they be used on websites.

Allied Vision Technologies GmbH 11/2015

Managing Director: Mr. Frank Grube
Tax ID: DE 184383113

Headquarters:

Taschenweg 2a
D-07646 Stadtroda, Germany
Tel.: +49 (0)36428 6770
Fax: +49 (0)36428 677-28
e-mail: info@alliedvision.com

Contents

List of Tables

Listings

1 Contact us

Connect with Allied Vision colleagues by function:
www.alliedvision.com/en/meta-header/contact

Find an Allied Vision office or distributor:
www.alliedvision.com/en/about-us/where-we-are.html

E-mail:
info@alliedvision.com (for commercial and general inquiries)
support@alliedvision.com (for technical assistance with Allied Vision products)

Telephone:
EMEA: +49 36428-677-0
The Americas: +1 978-225-2030
Asia-Pacific: +65 6634-9027
China: +86 (21) 64861133

Headquarters:
Allied Vision Technologies GmbH
Taschenweg 2a, 07646 Stadtroda, Germany
Tel: +49 (36428) 677-0 Fax +49 (36428) 677-24
President/CEO: Frank Grube | Registration Office: AG Jena HRB 208962

2 Introduction

2.1 Document history

Version	Date	Changes
1.0	2012-11-16	Initial version
1.1	2013-03-05	Minor corrections, added info about what functions can be called in which callback
1.2	2013-06-18	Small corrections, layout changes
1.3	2014-07-10	Appended the function reference, re-structured and made corrections
1.4	2015-11-10	Corrected GigE events, added USB compatibility, renamed several Vimba components and documents ("AVT" no longer in use), links to new Allied Vision website

2.2 Conventions used in this manual

To give this manual an easily understood layout and to emphasize important information, the following typographical styles and symbols are used:

2.2.1 Styles

Style	Function	Example
Bold	Programs, inputs or highlighting important things	**bold**
Courier	Code listings etc.	`Input`
Upper case	Constants	CONSTANT
Italics	Modes, fields, features	*Mode*
Blue and/or parentheses	Links	(Link)

2.2.2 Symbols

Note

This symbol highlights important information.

Caution

This symbol highlights important instructions. You have to follow these instructions to avoid malfunctions.

This symbol highlights URLs for further information. The URL itself is shown in blue.
Example: http://www.alliedvision.com

3 General aspects of the API

The Vimba C++ API is an object-oriented C++ API for enabling programmers to interact with Allied Vision cameras independent of the interface technology (Gigabit Ethernet, USB, 1394). It utilizes GenICam transport layer modules to connect to the various camera interfaces (1394, Gigabit Ethernet) and is therefore generic in terms of camera interfaces.

The C++ API is a sophisticated API with an elaborate class architecture and consequent use of shared pointers.

The entry point to Vimba C++ API is the `VimbaSystem` singleton. The `VimbaSystem` class allows both to control the API's behavior and to query for interfaces and cameras.

Note

The Vimba Manual contains a description of the API concepts.

Vimba API makes intense use of shared pointers to ease object lifetime and memory allocation. Since some C++ runtime libraries don't provide them, this Vimba API is equipped with an own implementation for Shared Pointers, which can be exchanged with your preferred shared pointer implementation very easily (see chapter Customizing shared pointer usage), for instance `std::shared_ptr`, `boost::shared_ptr`, or `QSharedPointer` from the Qt library.

For both compiling the Vimba C++ API and linking against it, a C++ compiler implementing the C++98 standard (ISO/IEC 14882:1998) is required.

4 API Usage

Note

The entry point to Vimba C++ API is the `VimbaSystem` singleton. To obtain a reference to it, call the static function `VimbaSystem::GetInstance`. All Vimba C++ classes reside in the namespace `AVT::VmbAPI`, so you might want to employ the using declaration `using AVT::VmbAPI`.

4.1 API Version

Even if new features are introduced to Vimba C++ API, your software remains backwards compatible. Use `VimbaSystem::QueryVersion` to check the version number of Vimba C++ API.

4.2 API Startup and Shutdown

In order to start and shut down Vimba C++ API, use these paired functions:

- `VimbaSystem::Startup` initializes Vimba API.
- `VimbaSystem::Shutdown` shuts down Vimba API (as soon as all observers have finished execution).

`VimbaSystem::Startup` and `VimbaSystem::Shutdown` must always be paired. Calling the pair several times within the same program is possible, but not recommended.

Successive calls of `VimbaSystem::Startup` or `VimbaSystem::Shutdown` are ignored and the first `VimbaSystem::Shutdown` after a `VimbaSystem::Startup` will close the API.

4.3 Shared Pointers

4.3.1 General aspects

A shared pointer is an object that wraps any regular pointer variable to control its lifetime. Besides wrapping the underlying raw pointer, it keeps track of the number of copies of itself. By doing so, it ensures that it will not release the wrapped raw pointer until its reference count (the number of copies) has dropped to zero. Though giving away the responsibility for deallocation, the programmer can still work on the very same objects.

Listing 1: Shared Pointers

```
1 {
2   // This declares an empty shared pointer that can wrap a pointer of
3   // type Camera
4   CameraPtr sp1;
5
6   // The reset member function tells the shared pointer to
7   // wrap the provided raw pointer
8   // sp1 now has a reference count of 1
9   sp1.reset( new Camera() );
```

```
10    {
11      // In this new scope we declare another shared pointer
12      CameraPtr sp2;
13
14      // By assigning sp1 to it the reference count of both (!) is set to 2
15      sp2 = sp1;
16    }
17    // When sp2 goes out of scope the reference count drops back to 1
18  }
19  // Now that sp1 has gone out of scope its reference count has dropped
20  // to 0 and it has released the underlying raw pointer on destruction
```

Unfortunately, shared pointers (or smart pointers in general) were not part of the C++ standard library until C++11. For example, the first version of Microsoft's C++ standard library implementation that supports shared pointers is included in Visual Studio 2010.

Because of the mentioned advantages, Vimba C++ API makes heavy use of shared pointers while not relying on a specific implementation. Although it is best practice to use the predefined shared pointer type, you can easily replace it with your own pointer type.

4.3.2 Customizing shared pointer usage

To use a custom shared pointer type in Vimba, follow these steps:

1. Add the define USER_SHARED_POINTER to your compiler settings
2. Add your shared pointer source files to the Vimba C++ API project
3. Define the macros and typedefs as described in the header UserSharedPointerDefines.h

The define USER_SHARED_POINTER tells Vimba to include a header file named UserSharedPointerDefines.h in which several typedefs and macros are defined.

Table 1 lists these macros covering the basic functionality that Vimba expects from any shared pointer. Since a shared pointer is a generic type, it requires a template parameter. That is what the various typedefs are for. For example, the `CameraPtr` is just an alias for `AVT::VmbAPI::shared_ptr<AVT::VmbAPI::Camera>`.

Macro	Example	Purpose
SP_DECL(T)	std::shared_ptr<T>	Declares a new shared pointer
SP_SET(sp, rawPtr)	sp.reset(rawPtr)	Tells an existing shared pointer to wrap the given raw pointer
SP_RESET(sp)	sp.reset()	Tells an existing shared pointer to decrease its reference count
SP_ISEQUAL(sp1, sp2)	(sp1 == sp2)	Checks the addresses of the wrapped raw pointers for equality
SP_ISNULL(sp)	(NULL == sp)	Checks the address of the wrapped raw pointer for NULL
SP_ACCESS(sp)	sp.get()	Returns the wrapped raw pointer
SP_DYN_CAST(sp, T)	std::dynamic_pointer_cast<T>(sp)	A dynamic cast of the pointer

Table 1: Basic functions of a shared pointer class

After you have completed these steps and have recompiled Vimba C++ API, Vimba is ready to use the provided shared pointer implementation without changing its behavior. Within your own application, you can employ your shared pointers as usual. Please note that your application and Vimba have to refer to the very same shared pointer type.

If you want your application to substitute its shared pointer type along with Vimba, feel free to utilize the macros listed in Table 1 in your application as well.

4.4 Listing available cameras

Note

For a quick start, see ListCameras example of the Vimba SDK.

`VimbaSystem::GetCameras` enumerates all cameras recognized by the underlying transport layers. With this command, the programmer can fetch the list of all connected camera objects. Before opening cameras, camera objects contain all static details of a physical camera that do not change throughout the object's lifetime such as:

- Camera ID
- Camera model
- Name or ID of the connected interface (for example, the network or 1394 adapter)

GigE cameras:
For GigE cameras, discovery has to be initiated by the host software. This is done automatically if you register a camera list observer with the `Vimba System` (of type `ICameraListObserver`). In this case, a call to `VimbaSystem::GetCameras` or `VimbaSystem::GetCameraByID` returns immediately.
If no camera list observer is registered, a call to `VimbaSystem::GetCameras` or `VimbaSystem::GetCameraByID` takes some time because the responses to the initiated discovery command must be waited for.

USB and 1394 cameras:
Changes to the plugged cameras are detected automatically. Consequently, any changes to the camera list are announced via discovery events, and the call to `VimbaSystem::GetCameras` returns immediately.

See Listing 2 for an example of **getting the camera list**.

Listing 2: Get Cameras

```
1  std::string name;
2  CameraPtrVector cameras;
3  VimbaSystem &system = VimbaSystem::GetInstance();
4
5  if ( VmbErrorSuccess == system.Startup() )
6  {
7    if ( VmbErrorSuccess == system.GetCameras( cameras ) )
8    {
9      for (   CameraPtrVector::iterator iter = cameras.begin();
10             cameras.end() != iter;
11             ++iter )
```

```
12        {
13          if ( VmbErrorSuccess == (*iter)->GetName( name ) )
14          {
15            std::cout << name << std::endl;
16          }
17        }
18      }
19    }
```

The Camera class provides the member functions listed in Table 2 to obtain information about a camera.

Function (returning VmbErrorType)	Purpose
GetID(std::string&) const	The unique ID
GetName(std::string&) const	The name
GetModel(std::string&) const	The model name
GetSerialNumber(std::string&) const	The serial number
GetPermittedAccess(VmbAccessModeType&) const	The mode to open the camera
GetInterfaceID(std::string&) const	The ID of the interface the camera is connected to

Table 2: Basic functions of the Camera class

Notifications of changed camera states

For being notified whenever a camera is detected, disconnected, or changes its open state, use VimbaSystem::RegisterCameraListObserver. This call registers a camera list observer (of type ICameraListObserver) with the Vimba System that gets executed on the according event. The observer function to be registered has to be of type ICameraListObserver*.

Note

VimbaSystem::Shutdown blocks until all callbacks have finished execution.

Caution

Functions that must **not** be called within your camera list observer:

- VimbaSystem::Startup
- VimbaSystem::Shutdown
- VimbaSystem::GetCameras
- VimbaSystem::GetCameraByID
- VimbaSystem::RegisterCameraListObserver
- VimbaSystem::UnregisterCameraListObserver
- Feature::SetValue
- Feature::RunCommand

4.5 Opening and closing a camera

A camera must be opened to control it and to capture images.

Call `Camera::Open` with the camera list entry of your choice, or use function `VimbaSystem::OpenCameraByID` with the ID of the camera. In both cases, also provide the desired access mode for the camera.
Vimba API provides several **access modes**:

- `VmbAccessModeFull` - read and write access. Use this mode to configure the camera features and to acquire images
- `VmbAccessModeConfig` - enables configuring the IP address of your GigE camera
- `VmbAccessModeRead` - read-only access.

An example for **opening a camera** retrieved from the camera list is shown in Listing 3.

Listing 3: Open Camera

```
1  CameraPtrVector cameras;
2  VimbaSystem &system = VimbaSystem::GetInstance();
3
4  if ( VmbErrorSuccess == system.Startup() )
5  {
6    if ( VmbErrorSuccess == system.GetCameras( cameras ) )
7    {
8      for (   CameraPtrVector::iterator iter = cameras.begin();
9              cameras.end() != iter;
10             ++iter )
11     {
12       if ( VmbErrorSuccess == (*iter)->Open( VmbAccessModeFull ) )
13       {
14         std::cout << "Camera opened" << std::endl;
15       }
16     }
17   }
18 }
```

Listing 4 shows how to **open a GigE camera by its IP address**.

Listing 4: Open Camera by IP

```
1  CameraPtr camera;
2  VimbaSystem &system = VimbaSystem::GetInstance();
3
4  if ( VmbErrorSuccess == system.Startup() )
5  {
6    if ( VmbErrorSuccess == system.OpenCameraByID(  "192.168.0.42",
7                                                    VmbAccessModeFull,
8                                                    camera ) )
9    {
10     std::cout << "Camera opened" << std::endl;
11   }
12 }
```

Listing 5 shows how to **close a camera** using `Camera::Close`.

Listing 5: Closing a camera

```
1  // the "camera" object points to an opened camera
2  if ( VmbErrorSuccess == camera.Close() )
3  {
4    std::cout << "Camera closed" << std::endl;
5  }
```

4.6 Accessing Features

Note

For a quick start, see ListFeatures example of the Vimba SDK.

GenICam-compliant features control and monitor various aspects of the drivers and cameras. For more details on features, see (if installed):

- GigE Features Reference (GigE camera features)
- USB Features Reference (USB camera features)
- Vimba 1394 TL Features Manual (1394 camera and TL features)
- Vimba Features Manual (Vimba System features)

There are several feature types which have type-specific properties and allow type-specific functionality. Vimba API provides its own set of access functions for each of these feature types.

Table 3 lists the Vimba API functions of the `Feature` class used to access feature values.

Feature Type	Set	Get	Range/Increment

`Enum`	`SetValue( string )`
	`SetValue( int )`
		`GetValue( string& )`
		`GetValue( int& )`
		`GetEntry( EnumEntry& )`
			`GetValues( StringVector& )`
			`GetValues( IntVector& )`
			`GetEntries( EntryVector& )`

`Int`	`SetValue(int )`	`GetValue( int& )`	`GetRange( int&, int& )` `GetIncrement( int& )`
`Float`	`SetValue( double )`	`GetValue( double& )`	`GetRange( double&, double& )`
`String`	`SetValue( string )`	`GetValue( string& )`	
`Bool`	`SetValue( bool )`	`GetValue( bool& )`	
`Command`	`RunCommand( )`	`IsCommandDone( bool& )`	
`Raw`	`SetValue( uchar )`	`GetValue( UcharVector& )`	

`Table 3:` Functions for reading and writing a Feature

With the member function `GetValue`, a feature's value can be queried.

With the member function `SetValue`, a feature's value can be set.

Integer and double features support `GetRange`. These functions return the minimum and maximum value that a feature can have. Integer features also support the `GetIncrement` function to query the step size of feature changes. Valid values for integer features are `min <= val <= min + [(max-min)/increment] * increment` (the maximum value might not be valid).

Enumeration features support `GetValues` that returns a vector of valid enumerations as strings or integers. These values can be used to set the feature according to the result of `IsValueAvailable`. If a non-empty vector is supplied, the original content is overwritten and the size of the vector is adjusted to fit all elements. An enumeration feature can also be used in a similar way as an integer feature.

Since not all the features are available all the time, the current accessibility of features may be queried via methods `IsReadable()` and `IsWritable()`, and the availability of Enum values may be queried with functions `IsValueAvailable( string )` or `IsValueAvailable( int )`.

With `Camera::GetFeatures`, you can list all features available for a camera. This list remains static while the camera is opened. The `Feature` class of the entries in this list also provides information about the features that always stay the same for this camera. Use the following member functions of class `Feature` to access them:

Function (returning `VmbErrorType`)	Purpose
`GetName( std::string& )`	Name of the feature
`GetDisplayName( std::string& )`	Name to display in GUI
`GetDataType( VmbFeatureDataType& )`	Data type of the feature. Gives information about the available functions for the feature. See table 3
`GetFlags( VmbFeatureFlagsType& )`	Static feature flags, containing information about the actions available for a feature and how changes might affect it. `Read`[1] and `Write`[1] flags determine whether get and set functions might succeed. `Volatile` features may change with every successive read. When writing `ModifyWrite` features, they will be adjusted to valid values
`GetCategory( std::string& )`	Category the feature belongs to, used for structuring the features
`GetPollingTime( VmbUint32_t& )`	The suggested time to poll the feature
`GetUnit( std::string& )`	The unit of the feature, if available
`GetRepresentation( std::string& )`	The scale to represent the feature, used as a hint for feature control
`GetVisibility( VmbFeatureVisibilityType& )`	The audience the feature is for
`GetToolTip( std::string& )`	Short description of the feature, used for bubble help
`GetDescription( std::string& )`	Description of the feature, used as extended explanation
`GetSFNCNamespace( std::string& )`	The SFNC namespace of the feature
`GetAffectedFeatures( FeaturePtrVector& )`	Features that change if the feature is changed
`GetSelectedFeatures( FeaturePtrVector& )`	Features that are selected by the feature

Table 4: Functions for accessing static properties of a Feature

[1] For the current availability, query methods `IsReadable()` and `IsWritable()`, and the availability of Enum values may be queried with functions `IsValueAvailable( std::string )` or `IsValueAvailable( int )`.

For an example of **reading a camera feature**, see Listing 6.

Listing 6: Reading a camera feature

```
FeaturePtr feature;
VmbInt64_t width;

if ( VmbErrorSuccess == camera->GetFeatureByName( "Width", feature )
{
  if ( VmbErrorSuccess == feature->GetValue( width ) )
  {
    std::out << width << std::endl;
  }
}
```

As an example for **writing features to a camera** and **running a command feature**, see Listing 7.

Listing 7: Writing features and running command features

```
FeaturePtr feature;

if ( VmbErrorSuccess == camera->GetFeatureByName( "AcquisitionMode", feature )
{
  if ( VmbErrorSuccess == feature->SetValue( "Continuous" ) )
  {
    if ( VmbErrorSuccess == camera->GetFeatureByName(   "AcquisitionStart",
                                                        feature ) )
    {
      if ( VmbErrorSuccess == feature->RunCommand() )
      {
        std::out << "Acquisition started" << std::endl;
      }
    }
  }
}
```

Table 5 introduces the basic features of all cameras. A feature has a name, a type, and access flags such as `read-permitted` and `write-permitted`.

Feature	Type	Access	Description
AcquisitionMode	Enumeration	R/W	The acquisition mode of the camera. Values: Continuous, SingleFrame, MultiFrame.
AcquisitionStart	Command		Start acquiring images.
AcquisitionStop	Command		Stop acquiring images.
PixelFormat	Enumeration	R/W	The image format. Possible values are e.g.: Mono8, RGB8Packed, YUV411Packed, BayerRG8, ...
Width	Uint32	R/W	Image width, in pixels.
Height	Uint32	R/W	Image height, in pixels.
PayloadSize	Uint32	R	Number of bytes in the camera payload, including the image.

Table 5: Basic features found on all cameras

To **get notified when a feature's value changes** use `Feature::RegisterObserver` (see chapter Using Events). The observer to be registered has to implement the interface `IFeatureObserver`. This interface declares the member function `FeatureChanged`. In the implementation of this function, you can react on updated feature values as it will get called by Vimba API on the according event.

Note

`VimbaSystem::Shutdown` blocks until all callbacks have finished execution.

Caution

Functions that must **not** be called within the feature observer:

- `VimbaSystem::Startup`
- `VimbaSystem::Shutdown`
- `VimbaSystem::GetCameras`
- `VimbaSystem::GetCameraByID`
- `VimbaSystem::RegisterCameraListObserver`
- `VimbaSystem::UnregisterCameraListObserver`
- `Feature::SetValue`
- `Feature::RunCommand`

4.7 Image Capture (API) and Acquisition (Camera)

Note

The Vimba Manual describes the principles of synchronous and asynchronous image acquisition.

Note

For a quick start, see SynchronousGrab, AsynchronousGrab, or VimbaViewer examples of the Vimba SDK.

4.7.1 Image Capture and Image Acquisition

Image capture and image acquisition are two independent operations: **Vimba API captures** images, the **camera acquires** images.

To obtain an image from your camera, setup Vimba API to capture images before starting the acquisition on the camera:

Note

Note that the C++ API provides convenience functions for standard applications. These functions perform several procedures in just one step. However, for complex applications with special requirements, manual programming as described here is still required.

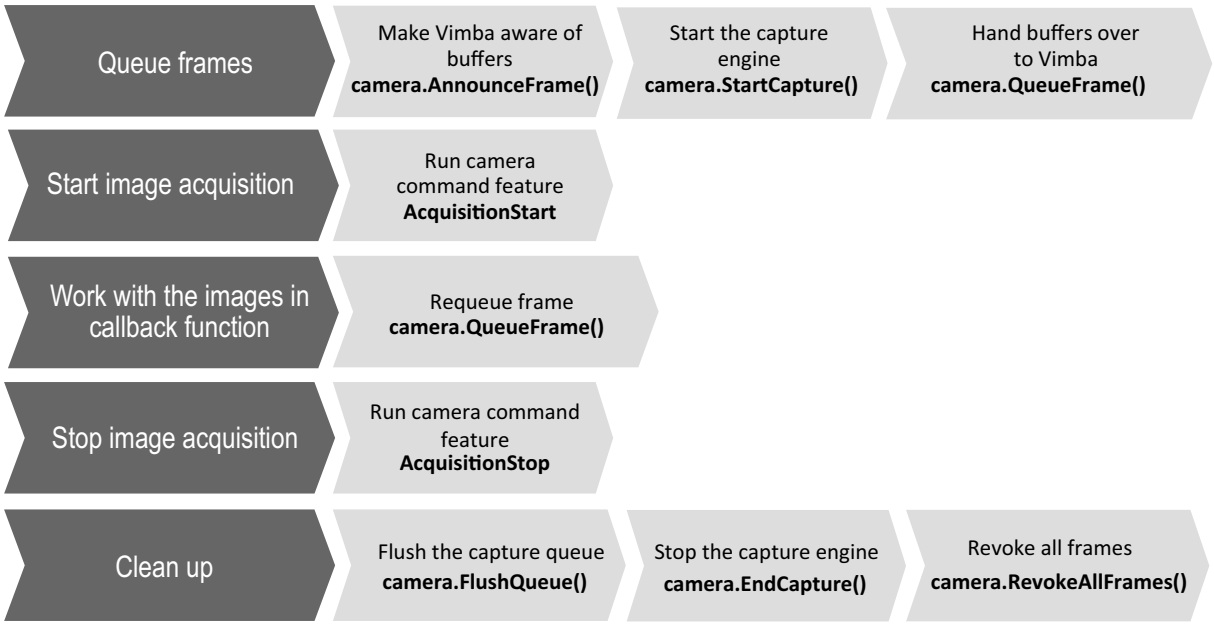

Figure 1: Typical asynchronous application using Vimba CPP

4.7.2 Image Capture

To enable image capture, frame buffers must be allocated, and the API must be prepared for incoming frames. This is done in convenience function `Camera::StartContinuousAcquisition` (`Camera::StopContinuousAcquisition` stops acquisition.). Please find below how to **asynchronously capture images** step by step:

1. Open the camera as described in chapter Opening and closing a camera
2. Query the necessary buffer size through the feature *PayloadSize* (A)[1]. Allocate frames of this size.(B)
3. Announce the frames (1).
4. Start the capture engine (2).
5. Queue the frame you have just created with `Camera::QueueFrame`, so that the buffer can be filled when the acquisition has started (3).
   The API is now ready. Start and stop image acquisition on the camera as described in chapter Image Acquisition.
6. Register a frame observer (C) that gets executed when capturing is complete.
   The frame observer has to be of type `IFrameObserver`. Within the frame observer, queue the frame again after you have processed it.
7. Call `Camera::FlushQueue` to cancel all frames on the queue. In case the API has done the memory allocation, this memory is not released until `RevokeAllFrames`, `RevokeFrame`, `EndCapture` or `Close` functions have been called.
8. Stop the capture engine with `Camera::EndCapture`.
9. Revoke the frames with `Camera::RevokeAllFrames` to clear the buffers.

To **synchronously capture images** (blocking your execution thread), follow these steps:

[1] The bracketed tokens in this chapter refer to Listing 8.

1. Open the camera as described in chapter Opening and closing a camera
2. How you proceed depends on the number of frames you need:
   - **A single frame:** Use `Camera::AcquireSingleImage` to receive an image frame.
   - **Multiple frames:** Use `Camera::AcquireMultipleImages` to receive several image frames (determined by the size of your vector of `FramePtrs`).

To assure correct continuous image capture, use at least two or three frames. The appropriate number of frames to be queued in your application depends on the frames per second the camera delivers and on the speed with which you are able to re-queue frames (also taking into consideration the operating system load). The image frames are filled in the same order in which they were queued.

Note

Always check that `Frame::GetReceiveStatus` returns `VmbFrameStatusComplete` when a frame is returned to ensure the data is valid.

Caution

Functions that must **not** be called within the frame observer:

- `VimbaSystem::Startup`
- `VimbaSystem::Shutdown`
- `VimbaSystem::OpenCameraByID`
- `Camera::Open`
- `Camera::Close`
- `Camera::AcquireSingleImage`
- `Camera::AcquireMultipleImages`
- `Camera::StartContinuousImageAcquisition`
- `Camera::StopContinuousImageAcquisition`
- `Camera::StartCapture`
- `Camera::EndCapture`
- `Camera::AnnounceFrame`
- `Camera::RevokeFrame`
- `Camera::RevokeAllFrames`

4.7.3 Image Acquisition

If you have decided to use one of the functions `Camera::AcquireSingleImage`, `Camera::AcquireMultipleImages`, or `Camera::StartContinuousImageAcquisition`, no further actions have to be taken.

Only if you have setup capture step by step as described in chapter Image Capture, you have to start image acquisition on your camera:

1. Set the feature *AcquisitionMode* (e.g. to *Continuous*).
2. Run the command *AcquisitionStart* (4).

To stop image acquisition, run command *AcquisitionStop*.

Listing 8 shows a **simplified streaming example** (without error handling).

Listing 8: Streaming

```
1  VmbErrorType err;    // Every Vimba function returns an error code that the
2                       // programmer should always check for VmbErrorSuccess
3  VimbaSystem &sys;    // A reference to the VimbaSystem singleton
4  CameraPtrVector cameras;     // A list of known cameras
5  FramePtrVector frames( 3 ); // A list of frames for streaming. We chose
6                              // to queue 3 frames.
7  IFrameObserverPtr pObserver( new MyFrameObserver() );    // Our implementation
8                                                           // of a frame observer
9  FeaturePtr pFeature;         // Any camera feature
10 VmbInt64_t nPLS;             // The payload size of one frame
11
12 sys = VimbaSystem::GetInstance();
13
14 err = sys.GetCameras( cameras );
15
16 err = cameras[0]->Open( VmbAccessModeFull );
17
18 err = cameras[0]->GetFeatureByName( "PayloadSize", pFeature );              (A)
19 err = pFeature->GetValue( nPLS )                                            (A)
20
21 for (   FramePtrVector::iterator iter = frames.begin();
22         frames.end() != iter;
23         ++iter )
24 {
25     ( *iter ).reset( new Frame( nPLS ) );                                   (B)
26     err = ( *iter )->RegisterObserver( pObserver ) );                       (C)
27     err = cameras[0]->AnnounceFrame( *iter  );                              (1)
28 }
29
30 err = StartCapture();                                                       (2)
31
32 for (   FramePtrVector::iterator iter = frames.begin();
33         frames.end() != iter;
34         ++iter )
35 {
36     err = cameras[0]->QueueFrame( *iter  );                                 (3)
37 }
38
39 err = GetFeatureByName( "AcquisitionStart", pFeature );                     (4)
40 err = pFeature->RunCommand();                                               (4)
```

4.8 Using Events

Events serve a multitude of purposes and can have several origins: The Vimba System, an Interface and cameras.

In Vimba, notifications are issued as a result to a feature invalidation of either its value or its state. Consequently, to get notified about any feature change, register an observer of the desired type (`ICameraListObserver`, `IInterfaceListObserver`, or `IFeatureObserver`) with the appropriate RegisterXXXObserver method (`RegisterCameraListObserver`, `RegisterInterfaceListObserver`, or `RegisterObserver`), which gets called if there is a change to that feature.

Three examples are listed in this chapter:

- Camera list notifications
- Tracking invalidations of features
- Explicit camera event features.

See Listing 9 for an example of being notified about **camera list changes**.

Listing 9: Getting notified about camera list changes

```
// 1. define observer that translates camera list changes to a Windows message
class CamObserver : public ICameraListObserver
{
...
public:
  void CameraListChanged( CameraPtr pCam, UpdateTriggerType reason )
  {
    if ( UpdateTriggerPluggedIn == reason || UpdateTriggerPluggedOut == reason )
    {
      CWinApp *pApp = AfxGetApp();
      if ( NULL != pApp )
      {
        CWnd *pMainWin = pApp->GetMainWnd();
        if ( NULL != pMainWin )
        {
          LRESULT b = pMainWin->PostMessage( WM_CAM_LIST_CHANGED, reason );
        }
      }
    }
  }
};

{
 VmbErrorType res;
 VimbaSystem &sys = VimbaSystem::GetInstance();
 FeaturePtr pFeature;

 // 2. Register the observer; automatic discovery for GigE is turned on
 res = sys.RegisterCameraListObserver( ICameraListObserverPtr( new CamObserver() ));
}
```

See Listing 10 for an example of being notified about **feature changes**.

Listing 10: Getting notified about feature changes

```
// 1. define observer
```

```
class WidthObserver : public IFeatureObserver
{
...
public:
  void FeatureChanged ( const FeaturePtr &feature )
  {
    if ( feature != NULL )
    {
      VmbError_t res;
      std::string strName("");

      res = feature->GetDisplayName(strName);
      std::cout << strName << " changed" << std:endl;
    }
  }
};

{
 ...
 // 2. register the observer for that event
 res = GetFeatureByName( "Width", pFeature );
 res = pFeature->RegisterFeatureObserver( IFeatureObserverPtr( new WidthObserver() ));

// as an example, binning is changed, so the observer will be run
 res = GetFeatureByName( "BinningHorizontal", pFeature );
 pFeature->SetValue(8);
}
```

Camera events are also handled with the same mechanism of feature invalidation. See Listing 11 for an example. For more details about camera events, see (if installed):

- GigE Features Reference (GigE camera features)
- USB Features Reference (USB camera features)
- Vimba 1394 TL Features Manual (1394 camera and TL features)
- Vimba Features Manual (Vimba System features)

Listing 11: Getting notified about camera events

```
// 1. define observer
class EventObserver : public IFeatureObserver
{
...
public:
  void FeatureChanged ( const FeaturePtr &feature )
  {
    if ( feature != NULL )
    {
      VmbError_t res;
      std::string strName("");

      res = feature->GetDisplayName(strName);
      std::cout "Event " << strName << " occurred" << std:endl;
    }
  }
};

{
```

```
20    ...
21    // 2. register the observer for the camera event
22    res = GetFeatureByName( "EventAcquisitionStart", pFeature );
23    res = pFeature->RegisterFeatureObserver( IFeatureObserverPtr( new EventObserver() ));
24
25    // 3. select "AcquisitionStart" event
26    res = GetFeatureByName( "EventSelector", pFeature );
27    res = pFeature->SetValue( "AcquisitionStart" );
28
29    // 4. switch on the event notification
30    res = GetFeatureByName( "EventNotification", pFeature );
31    res = pFeature->SetValue( "On" );
32 }
```

4.9 Additional configuration: Listing Interfaces

`VimbaSystem::GetInterfaces` will enumerate all Interfaces (GigE, USB, or 1394 adapters) recognized by the underlying transport layers.
See Listing 12 for an example.

Listing 12: Get Interfaces

```
std::string name;
InterfacePtrVector interfaces;
VimbaSystem &system = VimbaSystem::GetInstance();

if ( VmbErrorSuccess == system.Startup() )
{
  if ( VmbErrorSuccess == system.GetInterfaces( interfaces ) )
  {
    for (   InterfacePtrVector::iterator iter = interfaces.begin();
            interfaces.end() != iter;
            ++iter )
    {
      if ( VmbErrorSuccess == (*iter)->GetName( name ) )
      {
        std::cout << name << std::endl;
      }
    }
  }
}
```

The Interface class provides the member functions to obtain information about an interface listed in Table 6.

Function (returning `VmbErrorType`)	Purpose
`GetID( std::string& ) const`	The unique ID
`GetName( std::string& ) const`	The name
`GetType( VmbInterfaceType& ) const`	The camera interface type
`GetSerialNumber( std::string& ) const`	The serial number (not in use)
`GetPermittedAccess( VmbAccessModeType& ) const`	The mode to open the interface

Table 6: Basic functions of Interface class

Static features that do not change throughout the object's lifetime such as `ID` and `Name` can be queried without having to open the interface.
To **get notified when an Interface is detected or disconnected**, use `VimbaSystem::RegisterInterfaceListObserver` (see Chapter Using Events). The observer to be registered has to implement the interface `IInterfaceListObserver`. This interface declares the member function `InterfaceListChanged`. In your implementation of this function, you can react on interfaces being plugged in or out as it will get called by Vimba API on the according event.

4.10 Troubleshooting

4.10.1 GigE cameras

Make sure to set the *PacketSize* feature of GigE cameras to a value supported by your network card. If you use more than one camera on one interface, the available bandwidth has to be shared between the cameras.

- *GVSPAdjustPacketSize* configures GigE cameras to use the largest possible packets.
- *StreamBytesPerSecond* enables to configure the individual bandwidth if multiple cameras are used.
- The maximum packet size might not be available on all connected cameras. Try to reduce the packet size.

Further readings:

The GigE Installation Manual provides detailed information on how to configure your system.

4.10.2 USB cameras

Under Windows, make sure the correct driver is applied. For more details, see Vimba Manual, chapter Vimba Driver Installer.

In order to achieve best performance, see the technical manual of your USB camera, chapter Troubleshooting:
http://www.alliedvision.com/en/support/technical-documentation.html

4.11 Error Codes

All Vimba API functions return an error code of type `VmbErrorType`, which, for the sake of simplicity and uniformity, are the same as for the underlying C API.
Typical errors are listed with each function in chapter Function reference. However, any of the error codes listed in Table 7 might be returned.

Error Code	Value	Description
`VmbErrorSuccess`	0	No error
`VmbErrorInternalFault`	-1	Unexpected fault in Vimba or driver
`VmbErrorApiNotStarted`	-2	Startup was not called before the current command
`VmbErrorNotFound`	-3	The designated instance (camera, feature etc.) cannot be found
`VmbErrorBadHandle`	-4	The given handle is not valid
`VmbErrorDeviceNotOpen`	-5	Device was not opened for usage
`VmbErrorInvalidAccess`	-6	Operation is invalid with the current access mode
`VmbErrorBadParameter`	-7	One of the parameters is invalid (usually an illegal pointer)
`VmbErrorStructSize`	-8	The given struct size is not valid for this version of the API
`VmbErrorMoreData`	-9	More data available in a string/list than space is provided
`VmbErrorWrongType`	-10	Wrong feature type for this access function
`VmbErrorInvalidValue`	-11	The value is not valid; either out of bounds or not an increment of the minimum
`VmbErrorTimeout`	-12	Timeout during wait
`VmbErrorOther`	-13	Other error
`VmbErrorResources`	-14	Resources not available (e.g. memory)
`VmbErrorInvalidCall`	-15	Call is invalid in the current context (e.g. callback)
`VmbErrorNoTL`	-16	No transport layers are found
`VmbErrorNotImplemented`	-17	API feature is not implemented
`VmbErrorNotSupported`	-18	API feature is not supported
`VmbErrorIncomplete`	-19	A multiple registers read or write is partially completed

Table 7: Error codes returned by Vimba

5 Function reference

In this chapter you can find a complete list of all methods of the following classes/interfaces: `VimbaSystem`, `Interface`, `FeatureContainer`, `IRegisterDevice`, `IInterfaceListObserver`, `ICameraListObserver`, `IFrameObserver`, `IFeatureObserver`, `ICameraFactory`, `Camera`, `Frame`, `Feature`, `EnumEntry` and `AncillaryData`.

Methods in this chapter are always described in the same way:

- The caption states the name of the function without parameters
- The first item is a brief description
- The parameters of the function are listed in a table (with type, name, and description)
- The return values or the returned type is listed
- Finally, a more detailed description about the function is given

5.1 VimbaSystem

5.1.1 GetInstance()

Returns a reference to the System singleton.

- **VimbaSystem&**

5.1.2 QueryVersion()

Retrieve the version number of VmbAPI.

	Type	Name	Description
out	`VmbVersionInfo_t&`	version	Reference to the struct where version information is copied

- **VmbErrorSuccess:** always returned

Note

This function can be called at any time, even before the API is initialized. All other version numbers may be queried via feature access

5.1.3 Startup()

Initialize the VmbAPI module.

- **VmbErrorSuccess:** If no error
- **VmbErrorInternalFault:** An internal fault occurred

Note

On successful return, the API is initialized; this is a necessary call. This method must be called before any other VmbAPI function is run.

5.1.4 Shutdown()

Perform a shutdown on the API module.

- **VmbErrorSuccess:** always returned

Note

This will free some resources and deallocate all physical resources if applicable.

5.1.5 GetInterfaces()

List all the interfaces currently visible to VmbAPI.

	Type	Name	Description
out	`InterfacePtrVector&`	interfaces	Vector of shared pointer to Interface object

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorStructSize:** The given struct size is not valid for this API version
- **VmbErrorMoreData:** More data were returned than space was provided
- **VmbErrorInternalFault:** An internal fault occurred

Note

All the interfaces known via a GenTL are listed by this command and filled into the vector provided. If the vector is not empty, new elements will be appended. Interfaces can be adapter cards or frame grabber cards, for instance.

5.1.6 GetInterfaceByID()

Gets a specific interface identified by an ID.

	Type	Name	Description
in	`const char*`	pID	The ID of the interface to get (returned by GetInterfaces())
out	`InterfacePtr&`	pInterface	Shared pointer to Interface object

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadParameter:** "pID" is NULL.
- **VmbErrorStructSize:** The given struct size is not valid for this API version
- **VmbErrorMoreData:** More data were returned than space was provided

Note

An interface known via a GenTL is listed by this command and filled into the pointer provided. Interface can be an adapter card or a frame grabber card, for instance.

5.1.7 OpenInterfaceByID()

Open an interface for feature access.

	Type	Name	Description
in	`const char*`	pID	The ID of the interface to open (returned by GetInterfaces())
out	`InterfacePtr&`	pInterface	A shared pointer to the interface

- **VmbErrorSuccess:** If no error

- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorNotFound:** The designated interface cannot be found
- **VmbErrorBadParameter:** "pID" is NULL.

Note

An interface can be opened if interface-specific control is required, such as I/O pins on a frame grabber card. Control is then possible via feature access methods.

5.1.8 GetCameras()

Retrieve a list of all cameras.

	Type	Name	Description
out	`CameraPtrVector&`	cameras	Vector of shared pointer to Camera object

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorStructSize:** The given struct size is not valid for this API version
- **VmbErrorMoreData:** More data were returned than space was provided

Note

A camera known via a GenTL is listed by this command and filled into the pointer provided.

5.1.9 GetCameraByID()

Gets a specific camera identified by an ID. The returned camera is still closed.

	Type	Name	Description
in	`const char*`	pID	The ID of the camera to get
out	`CameraPtr&`	pCamera	Shared pointer to camera object

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadParameter:** "pID" is NULL.
- **VmbErrorStructSize:** The given struct size is not valid for this API version
- **VmbErrorMoreData:** More data were returned than space was provided

Note

A camera known via a GenTL is listed by this command and filled into the pointer provided. Only static properties of the camera can be fetched until the camera has been opened. "pID" might be one of the following: "169.254.12.13" for an IP address, "000F314C4BE5" for a MAC address or "1234567890" for a plain serial number.

5.1.10 OpenCameraByID()

Gets a specific camera identified by an ID. The returned camera is already open.

	Type	Name	Description
in	`const char*`	pID	The unique ID of the camera to get
in	`VmbAccessModeType`	eAccessMode	The requested access mode
out	`CameraPtr&`	pCamera	A shared pointer to the camera

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorNotFound:** The designated interface cannot be found
- **VmbErrorBadParameter:** "pID" is NULL.

Note

A camera can be opened if camera-specific control is required, such as I/O pins on a frame grabber card. Control is then possible via feature access methods. "pID" might be one of the following: "169.254.12.13" for an IP address, "000F314C4BE5" for a MAC address or "1234567890" for a plain serial number.

5.1.11 RegisterCameraListObserver()

Registers an instance of camera observer whose CameraListChanged() method gets called as soon as a camera is plugged in, plugged out, or changes its access status

	Type	Name	Description
in	`const ICameraListObserverPtr&`	pObserver	A shared pointer to an object derived from ICameraListObserver

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pObserver" is NULL.
- **VmbErrorInvalidCall:** If the very same observer is already registered

5.1.12 UnregisterCameraListObserver()

Unregisters a camera observer

	Type	Name	Description
in	`const ICameraListObserverPtr&`	pObserver	A shared pointer to an object derived from ICameraListObserver

- **VmbErrorSuccess:** If no error
- **VmbErrorNotFound:** If the observer is not registered
- **VmbErrorBadParameter:** "pObserver" is NULL.

5.1.13 RegisterInterfaceListObserver()

Registers an instance of interface observer whose InterfaceListChanged() method gets called as soon as an interface is plugged in, plugged out, or changes its access status

	Type	Name	Description
in	`const IInterfaceListObserverPtr&`	pObserver	A shared pointer to an object derived from IInterfaceListObserver

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pObserver" is NULL.
- **VmbErrorInvalidCall:** If the very same observer is already registered

5.1.14 UnregisterInterfaceListObserver()

Unregisters an interface observer

	Type	Name	Description
in	`const IInterfaceListObserverPtr&`	pObserver	A shared pointer to an object derived from IInterfaceListObserver

- **VmbErrorSuccess:** If no error
- **VmbErrorNotFound:** If the observer is not registered
- **VmbErrorBadParameter:** "pObserver" is NULL.

5.1.15 RegisterCameraFactory()

Registers an instance of camera factory. When a custom camera factory is registered, all instances of type camera will be set up accordingly.

	Type	Name	Description
in	`const ICameraFactoryPtr&`	pCameraFactory	A shared pointer to an object derived from ICameraFactory

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pCameraFactory" is NULL.

5.1.16 UnregisterCameraFactory()

Unregisters the camera factory. After unregistering the default camera class is used.

- **VmbErrorSuccess:** If no error

5.2 Interface

5.2.1 Open()

Open an interface handle for feature access.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorNotFound:** The designated interface cannot be found

Note

An interface can be opened if interface-specific control is required, such as I/O pins on a frame grabber card. Control is then possible via feature access methods.

5.2.2 Close()

Close an interface.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The handle is not valid

5.2.3 GetID()

Gets the ID of an interface.

	Type	Name	Description
out	`std::string&`	interfaceID	The ID of the interface

- **VmbErrorSuccess:** If no error

Note

This information remains static throughout the object's lifetime

5.2.4 GetType()

Gets the type, e.g. FireWire, GigE or USB of an interface.

	Type	Name	Description
out	`VmbInterfaceType&`	type	The type of the interface

- **VmbErrorSuccess:** If no error

Note

This information remains static throughout the object's lifetime

5.2.5 GetName()

Gets the name of an interface.

	Type	Name	Description
out	`std::string&`	name	The name of the interface

- **VmbErrorSuccess:** If no error

5.2.6 GetSerialNumber()

Gets the serial number of an interface.

	Type	Name	Description
out	`std::string&`	serialNumber	The serial number of the interface

- **VmbErrorSuccess:** If no error

5.2.7 GetPermittedAccess()

Gets the access mode of an interface.

	Type	Name	Description
out	`VmbAccessModeType&`	accessMode	The possible access mode of the interface

- **VmbErrorSuccess:** If no error

5.3 FeatureContainer

5.3.1 FeatureContainer constructor

Creates an instance of class FeatureContainer

5.3.2 FeatureContainer destructor

Destroys an instance of class FeatureContainer

5.3.3 GetFeatureByName()

Gets one particular feature of a feature container (e.g. a camera)

	Type	Name	Description
in	`const char*`	name	The name of the feature to get
out	`FeaturePtr&`	pFeature	The queried feature

- **VmbErrorSuccess:** If no error
- **VmbErrorDeviceNotOpen:** Base feature class (e.g. Camera) was not opened.
- **VmbErrorBadParameter:** "name" is NULL.

5.3.4 GetFeatures()

Gets all features of a feature container (e.g. a camera)

	Type	Name	Description
out	`FeaturePtrVector&`	features	The container for all queried features

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "features" is empty.

Note

Once queried, this information remains static throughout the object's lifetime

5.4 IRegisterDevice

5.4.1 ReadRegisters()

Reads one or more registers consecutively. The number of registers to be read is determined by the number of provided addresses.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
out	`Uint64Vector&`	buffer	The returned data as vector

- **VmbErrorSuccess:** If all requested registers have been read
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been read. See overload ReadRegisters(const Uint64Vector&, Uint64Vector&, VmbUint32_t&).

5.4.2 ReadRegisters()

Same as ReadRegisters(const Uint64Vector&, Uint64Vector&), but returns the number of successful read operations in case of an error.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
out	`Uint64Vector&`	buffer	The returned data as vector
out	`VmbUint32_t&`	completedReads	The number of successfully read registers

- **VmbErrorSuccess:** If all requested registers have been read
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been read.

5.4.3 WriteRegisters()

Writes one or more registers consecutively. The number of registers to be written is determined by the number of provided addresses.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
in	`const Uint64Vector&`	buffer	The data to write as vector

- **VmbErrorSuccess:** If all requested registers have been written
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been written. See overload WriteRegisters(const Uint64Vector&, const Uint64Vector&, VmbUint32_t&).

5.4.4 WriteRegisters()

Same as WriteRegisters(const Uint64Vector&, const Uint64Vector&), but returns the number of successful write operations in case of an error VmbErrorIncomplete.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
in	`const Uint64Vector&`	buffer	The data to write as vector
out	`VmbUint32_t&`	completedWrites	The number of successfully read registers

- **VmbErrorSuccess:** If all requested registers have been written
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been written.

5.4.5 ReadMemory()

Reads a block of memory. The number of bytes to read is determined by the size of the provided buffer.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to read from
out	`UcharVector&`	buffer	The returned data as vector

- **VmbErrorSuccess:** If all requested bytes have been read
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been read. See overload ReadMemory(const VmbUint64_t&, UcharVector&, VmbUint32_t&).

5.4.6 ReadMemory()

Same as ReadMemory(const Uint64Vector&, UcharVector&), but returns the number of bytes successfully read in case of an error VmbErrorIncomplete.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to read from
out	`UcharVector&`	buffer	The returned data as vector
out	`VmbUint32_t&`	sizeComplete	The number of successfully read bytes

- **VmbErrorSuccess:** If all requested bytes have been read
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been read.

5.4.7 WriteMemory()

Writes a block of memory. The number of bytes to write is determined by the size of the provided buffer.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to write to
in	`const UcharVector&`	buffer	The data to write as vector

- **VmbErrorSuccess:** If all requested bytes have been written
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been written. See overload WriteMemory(const VmbUint64_t&, const UcharVector&, VmbUint32_t&).

5.4.8 WriteMemory()

Same as WriteMemory(const Uint64Vector&, const UcharVector&), but returns the number of bytes successfully written in case of an error VmbErrorIncomplete.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to write to
in	`const UcharVector&`	buffer	The data to write as vector
out	`VmbUint32_t&`	sizeComplete	The number of successfully written bytes

- **VmbErrorSuccess:** If all requested bytes have been written
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been written.

5.5 IInterfaceListObserver

5.5.1 InterfaceListChanged()

The event handler function that gets called whenever an IInterfaceListObserver is triggered.

	Type	Name	Description
out	`InterfacePtr`	pInterface	The interface that triggered the event
out	`UpdateTriggerType`	reason	The reason why the callback routine was triggered

5.5.2 IInterfaceListObserver destructor

Destroys an instance of class IInterfaceListObserver

5.6 ICameraListObserver

5.6.1 CameraListChanged()

The event handler function that gets called whenever an ICameraListObserver is triggered. This occurs most likely when a camera was plugged in or out.

	Type	Name	Description
out	`CameraPtr`	pCam	The camera that triggered the event
out	`UpdateTriggerType`	reason	The reason why the callback routine was triggered (e.g., a new camera was plugged in)

5.6.2 ICameraListObserver destructor

Destroys an instance of class ICameraListObserver

5.7 IFrameObserver

5.7.1 FrameReceived()

The event handler function that gets called whenever a new frame is received

	Type	Name	Description
in	`const FramePtr`	pFrame	The frame that was received

5.7.2 IFrameObserver destructor

Destroys an instance of class IFrameObserver

5.8 IFeatureObserver

5.8.1 FeatureChanged()

The event handler function that gets called whenever a feature has changed

	Type	Name	Description
in	`const FeaturePtr&`	pFeature	The frame that has changed

5.8.2 IFeatureObserver destructor

Destroys an instance of class IFeatureObserver

5.9 ICameraFactory

5.9.1 CreateCamera()

Factory method to create a camera that extends the Camera class

	Type	Name	Description
in	`const char*`	pCameraID	The ID of the camera
in	`const char*`	pCameraName	The name of the camera
in	`const char*`	pCameraModel	The model name of the camera
in	`const char*`	pCameraSerialNumber	The serial number of the camera
in	`const char*`	pInterfaceID	The ID of the interface the camera is connected to
in	`VmbInterfaceType`	interfaceType	The type of the interface the camera is connected to
in	`const char*`	pInterfaceName	The name of the interface
in	`const char*`	pInterfaceSerialNumber	The serial number of the interface
in	`VmbAccessModeType`	interfacePermittedAccess	The access privileges for the interface

Note

The ID of the camera may be, among others, one of the following: "169.254.12.13", "000f31000001", a plain serial number: "1234567890", or the device ID of the underlying transport layer.

5.9.2 ICameraFactory destructor

Destroys an instance of class Camera

5.10 Camera

5.10.1 Camera constructor

Creates an instance of class Camera

	Type	Name	Description
in	`const char*`	pID	The ID of the camera
in	`const char*`	pName	The name of the camera
in	`const char*`	pModel	The model name of the camera
in	`const char*`	pSerialNumber	The serial number of the camera
in	`const char*`	pInterfaceID	The ID of the interface the camera is connected to
in	`VmbInterfaceType`	interfaceType	The type of the interface the camera is connected to

Note

The ID of the camera may be, among others, one of the following: "169.254.12.13", "000f31000001", a plain serial number: "1234567890", or the device ID of the underlying transport layer.

5.10.2 Camera destructor

Destroys an instance of class Camera

Note

Destroying a camera implicitly closes it beforehand.

5.10.3 Open()

Opens the specified camera.

	Type	Name	Description
in	`VmbAccessMode_t`	accessMode	Access mode determines the level of control you have on the camera

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorNotFound:** The designated camera cannot be found
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode

Note

A camera may be opened in a specific access mode. This mode determines the level of control you have on a camera.

5.10.4 Close()

Closes the specified camera.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command

Note

Depending on the access mode this camera was opened in, events are killed, callbacks are unregistered, the frame queue is cleared, and camera control is released.

5.10.5 GetID()

Gets the ID of a camera.

	Type	Name	Description
out	`std::string&`	cameraID	The ID of the camera

- **VmbErrorSuccess:** If no error

5.10.6 GetName()

Gets the name of a camera.

	Type	Name	Description
out	`std::string&`	name	The name of the camera

- **VmbErrorSuccess:** If no error

5.10.7 GetModel()

Gets the model name of a camera.

	Type	Name	Description
out	`std::string&`	model	The model name of the camera

- **VmbErrorSuccess:** If no error

5.10.8 GetSerialNumber()

Gets the serial number of a camera.

	Type	Name	Description
out	`std::string&`	serialNumber	The serial number of the camera

- **VmbErrorSuccess:** If no error

5.10.9 GetInterfaceID()

Gets the interface ID of a camera.

	Type	Name	Description
out	`std::string&`	interfaceID	The interface ID of the camera

- **VmbErrorSuccess:** If no error

5.10.10 GetInterfaceType()

Gets the type of the interface the camera is connected to. And therefore the type of the camera itself.

	Type	Name	Description
out	`VmbInterfaceType&`	interfaceType	The interface type of the camera

- **VmbErrorSuccess:** If no error

5.10.11 GetPermittedAccess()

Gets the access modes of a camera.

	Type	Name	Description
out	`VmbAccessModeType&`	permittedAccess	The possible access modes of the camera

- **VmbErrorSuccess:** If no error

5.10.12 ReadRegisters()

Reads one or more registers consecutively. The number of registers to read is determined by the number of provided addresses.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
out	`Uint64Vector&`	buffer	The returned data as vector

- **VmbErrorSuccess:** If all requested registers have been read
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been read. See overload ReadRegisters(const Uint64Vector&, Uint64Vector&, VmbUint32_t&).

5.10.13 ReadRegisters()

Same as ReadRegisters(const Uint64Vector&, Uint64Vector&), but returns the number of successful read operations in case of an error.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
out	`Uint64Vector&`	buffer	The returned data as vector
out	`VmbUint32_t&`	completedReads	The number of successfully read registers

- **VmbErrorSuccess:** If all requested registers have been read
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been read.

5.10.14 WriteRegisters()

Writes one or more registers consecutively. The number of registers to write is determined by the number of provided addresses.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
in	`const Uint64Vector&`	buffer	The data to write as vector

- **VmbErrorSuccess:** If all requested registers have been written
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been written. See overload WriteRegisters(const Uint64Vector&, const Uint64Vector&, VmbUint32_t&).

5.10.15 WriteRegisters()

Same as WriteRegisters(const Uint64Vector&, const Uint64Vector&), but returns the number of successful write operations in case of an error.

	Type	Name	Description
in	`const Uint64Vector&`	addresses	A list of register addresses
in	`const Uint64Vector&`	buffer	The data to write as vector
out	`VmbUint32_t&`	completedWrites	The number of successfully read registers

- **VmbErrorSuccess:** If all requested registers have been written
- **VmbErrorBadParameter:** Vectors "addresses" and/or "buffer" are empty.
- **VmbErrorIncomplete:** If at least one, but not all registers have been written.

5.10.16 ReadMemory()

Reads a block of memory. The number of bytes to read is determined by the size of the provided buffer.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to read from
out	`UcharVector&`	buffer	The returned data as vector

- **VmbErrorSuccess:** If all requested bytes have been read
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been read. See overload ReadMemory(const VmbUint64_t&, UcharVector&, VmbUint32_t&).

5.10.17 ReadMemory()

Same as ReadMemory(const Uint64Vector&, UcharVector&), but returns the number of bytes successfully read in case of an error VmbErrorIncomplete.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to read from
out	`UcharVector&`	buffer	The returned data as vector
out	`VmbUint32_t&`	completeReads	The number of successfully read bytes

- **VmbErrorSuccess:** If all requested bytes have been read
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been read.

5.10.18 WriteMemory()

Writes a block of memory. The number of bytes to write is determined by the size of the provided buffer.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to write to
in	`const UcharVector&`	buffer	The data to write as vector

- **VmbErrorSuccess:** If all requested bytes have been written
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been written. See overload WriteMemory(const VmbUint64_t&, const UcharVector&, VmbUint32_t&).

5.10.19 WriteMemory()

Same as WriteMemory(const Uint64Vector&, const UcharVector&), but returns the number of bytes successfully written in case of an error VmbErrorIncomplete.

	Type	Name	Description
in	`const VmbUint64_t&`	address	The address to write to
in	`const UcharVector&`	buffer	The data to write as vector
out	`VmbUint32_t&`	sizeComplete	The number of successfully written bytes

- **VmbErrorSuccess:** If all requested bytes have been written
- **VmbErrorBadParameter:** Vector "buffer" is empty.
- **VmbErrorIncomplete:** If at least one, but not all bytes have been written.

5.10.20 AcquireSingleImage()

Gets one image synchronously.

	Type	Name	Description
out	`FramePtr&`	pFrame	The frame that gets filled
in	`VmbUint32_t`	timeout	The time to wait until the frame got filled

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pFrame" is NULL.
- **VmbErrorTimeout:** Call timed out

5.10.21 AcquireMultipleImages()

Gets a certain number of images synchronously.

	Type	Name	Description
out	`FramePtrVector&`	frames	The frames that get filled
in	`VmbUint32_t`	timeout	The time to wait until one frame got filled

Note

The size of the frame vector determines the number of frames to use.

- **VmbErrorSuccess:** If no error
- **VmbErrorInternalFault:** Filling all the frames was not successful.
- **VmbErrorBadParameter:** Vector "frames" is empty.

5.10.22 AcquireMultipleImages(FramePtrVector&, VmbUint32_t), but returns the number of frames that were filled completely.

Same as AcquireMultipleImages()

	Type	Name	Description
out	`FramePtrVector&`	frames	The frames that get filled
in	`VmbUint32_t`	timeout	The time to wait until one frame got filled
out	`VmbUint32_t&`	numFramesCompleted	The number of frames that were filled completely

Note

The size of the frame vector determines the number of frames to use. On return, "numFramesCompleted" holds the number of frames actually filled.

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** Vector "frames" is empty.

5.10.23 StartContinuousImageAcquisition()

Starts streaming and allocates the needed frames

	Type	Name	Description
in	`int`	bufferCount	The number of frames to use
out	`const IFrameObserverPtr&`	pObserver	The observer to use on arrival of new frames

- **VmbErrorSuccess:** If no error
- **VmbErrorDeviceNotOpen:** The camera has not been opened before
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode

5.10.24 StopContinuousImageAcquisition()

Stops streaming and deallocates the needed frames

5.10.25 AnnounceFrame()

Announces a frame to the API that may be queued for frame capturing later.

	Type	Name	Description
in	`const FramePtr&`	pFrame	Shared pointer to a frame to announce

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid
- **VmbErrorBadParameter:** "pFrame" is NULL.
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API

Note

Allows some preparation for frames like DMA preparation depending on the transport layer. The order in which the frames are announced is not taken in consideration by the API.

5.10.26 RevokeFrame()

Revoke a frame from the API.

	Type	Name	Description
in	`const FramePtr&`	pFrame	Shared pointer to a frame that is to be removed from the list of announced frames

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given frame pointer is not valid
- **VmbErrorBadParameter:** "pFrame" is NULL.
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API

Note

The referenced frame is removed from the pool of frames for capturing images.

5.10.27 RevokeAllFrames()

Revoke all frames assigned to this certain camera.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid

5.10.28 QueueFrame()

Queues a frame that may be filled during frame capturing.

	Type	Name	Description
in	`const FramePtr&`	pFrame	A shared pointer to a frame

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given frame is not valid
- **VmbErrorBadParameter:** "pFrame" is NULL.
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API
- **VmbErrorInvalidCall:** StopContinuousImageAcquisition is currently running in another thread

Note

The given frame is put into a queue that will be filled sequentially. The order in which the frames are filled is determined by the order in which they are queued. If the frame was announced with AnnounceFrame() before, the application has to ensure that the frame is also revoked by calling RevokeFrame() or RevokeAll() when cleaning up.

5.10.29 FlushQueue()

Flushes the capture queue.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid

Note

All the currently queued frames will be returned to the user, leaving no frames in the input queue. After this call, no frame notification will occur until frames are queued again.

5.10.30 StartCapture()

Prepare the API for incoming frames from this camera.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid
- **VmbErrorDeviceNotOpen:** Camera was not opened for usage
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode

5.10.31 EndCapture()

Stop the API from being able to receive frames from this camera.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid

Note

Consequences of VmbCaptureEnd(): - The frame queue is flushed - The frame callback will not be called any more

5.11 Frame

5.11.1 Frame constructor

Creates an instance of class Frame of a certain size

	Type	Name	Description
in	`VmbInt64_t`	bufferSize	The size of the underlying buffer

5.11.2 Frame constructor

Creates an instance of class Frame with the given user buffer of the given size

	Type	Name	Description
in	`VmbUchar_t*`	pBuffer	A pointer to an allocated buffer
in	`VmbInt64_t`	bufferSize	The size of the underlying buffer

5.11.3 Frame destructor

Destroys an instance of class Frame

5.11.4 RegisterObserver()

Registers an observer that will be called whenever a new frame arrives

	Type	Name	Description
in	`const IFrameObserverPtr&`	pObserver	An object that implements the IObserver interface

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pObserver" is NULL.
- **VmbErrorResources:** The observer was in use

Note

As new frames arrive, the observer's FrameReceived method will be called. Only one observer can be registered.

5.11.5 UnregisterObserver()

Unregisters the observer that was called whenever a new frame arrived

5.11.6 GetAncillaryData()

Returns the part of a frame that describes the chunk data as an object

	Type	Name	Description
out	`AncillaryDataPtr&`	pAncillaryData	The wrapped chunk data

- **VmbErrorSuccess:** If no error
- **VmbErrorNotFound:** No chunk data present

5.11.7 GetAncillaryData()

Returns the part of a frame that describes the chunk data as an object

	Type	Name	Description
out	`ConstAncillaryDataPtr&`	pAncillaryData	The wrapped chunk data

- **VmbErrorSuccess:** If no error
- **VmbErrorNotFound:** No chunk data present

5.11.8 GetBuffer()

Returns the complete buffer including image and chunk data

	Type	Name	Description
out	`VmbUchar_t*`	pBuffer	A pointer to the buffer

- **VmbErrorSuccess:** If no error

5.11.9 GetBuffer()

Returns the complete buffer including image and chunk data

	Type	Name	Description
out	`const VmbUchar_t*`	pBuffer	A pointer to the buffer

- **VmbErrorSuccess:** If no error

5.11.10 GetImage()

Returns only the image data

	Type	Name	Description
out	`VmbUchar_t*`	pBuffer	A pointer to the buffer

- **VmbErrorSuccess:** If no error

5.11.11 GetImage()

Returns only the image data

	Type	Name	Description
out	`const VmbUchar_t*`	pBuffer	A pointer to the buffer

- **VmbErrorSuccess:** If no error

5.11.12 GetReceiveStatus()

Returns the receive status of a frame

	Type	Name	Description
out	`VmbFrameStatusType&`	status	The receive status

- **VmbErrorSuccess:** If no error

5.11.13 GetImageSize()

Returns the memory size of the image

	Type	Name	Description
out	`VmbUint32_t&`	imageSize	The size in bytes

- **VmbErrorSuccess:** If no error

5.11.14 GetAncillarySize()

Returns memory size of the chunk data

	Type	Name	Description
out	`VmbUint32_t&`	ancillarySize	The size in bytes

- **VmbErrorSuccess:** If no error

5.11.15 GetBufferSize()

Returns the memory size of the frame buffer holding both the image data and the ancillary data

	Type	Name	Description
out	`VmbUint32_t&`	bufferSize	The size in bytes

- **VmbErrorSuccess:** If no error

5.11.16 GetPixelFormat()

Returns the GenICam pixel format

	Type	Name	Description
out	`VmbPixelFormatType&`	pixelFormat	The GenICam pixel format

- **VmbErrorSuccess:** If no error

5.11.17 GetWidth()

Returns the width of the image

	Type	Name	Description
out	`VmbUint32_t&`	width	The width in pixels

- **VmbErrorSuccess:** If no error

5.11.18 GetHeight()

Returns the height of the image

	Type	Name	Description
out	`VmbUint32_t&`	height	The height in pixels

- **VmbErrorSuccess:** If no error

5.11.19 GetOffsetX()

Returns the x offset of the image

	Type	Name	Description
out	`VmbUint32_t&`	offsetX	The x offset in pixels

- **VmbErrorSuccess:** If no error

5.11.20 GetOffsetY()

Returns the y offset of the image

	Type	Name	Description
out	`VmbUint32_t&`	offsetY	The y offset in pixels

- **VmbErrorSuccess:** If no error

5.11.21 GetFrameID()

Returns the frame ID

	Type	Name	Description
out	`VmbUint64_t&`	frameID	The frame ID

- **VmbErrorSuccess:** If no error

5.11.22 GetTimeStamp()

Returns the time stamp

	Type	Name	Description
out	`VmbUint64_t&`	timestamp	The time stamp

- **VmbErrorSuccess:** If no error

5.12 Feature

5.12.1 GetValue()

Queries the value of a feature of type VmbInt64

	Type	Name	Description
out	`VmbInt64_t&`	value	The feature's value

5.12.2 GetValue()

Queries the value of a feature of type double

	Type	Name	Description
out	`double&`	value	The feature's value

5.12.3 GetValue()

Queries the value of a feature of type string

	Type	Name	Description
out	`std::string&`	value	The feature's value

Note

When an empty string is returned, its size indicates the maximum length

5.12.4 GetValue()

Queries the value of a feature of type bool

	Type	Name	Description
out	`bool&`	value	The feature's value

5.12.5 GetValue()

Queries the value of a feature of type UcharVector

	Type	Name	Description
out	`UcharVector&`	value	The feature's value

5.12.6 GetValue()

Queries the value of a feature of type const UcharVector

	Type	Name	Description
out	`UcharVector&`	value	The feature's value
out	`VmbUint32_t&`	sizeFilled	The number of actually received values

5.12.7 GetValues()

Queries the values of a feature of type Int64Vector

	Type	Name	Description
out	`Int64Vector&`	values	The feature's values

5.12.8 GetValues()

Queries the values of a feature of type StringVector

	Type	Name	Description
out	`StringVector&`	values	The feature's values

5.12.9 GetEntry()

Queries a single enum entry of a feature of type Enumeration

	Type	Name	Description
out	`EnumEntry&`	entry	An enum feature's enum entry
in	`const char*`	pEntryName	The name of the enum entry

5.12.10 GetEntries()

Queries all enum entries of a feature of type Enumeration

	Type	Name	Description
out	`EnumEntryVector&`	entries	An enum feature's enum entries

5.12.11 GetRange()

Queries the range of a feature of type double

	Type	Name	Description
out	`double&`	minimum	The feature's min value
out	`double&`	maximum	The feature's max value

5.12.12 GetRange()

Queries the range of a feature of type VmbInt64

	Type	Name	Description
out	`VmbInt64_t&`	minimum	The feature's min value
out	`VmbInt64_t&`	maximum	The feature's max value

5.12.13 SetValue()

Sets the value of a feature of type VmbInt32

	Type	Name	Description
in	`const VmbInt32_t&`	value	The feature's value

5.12.14 SetValue()

Sets the value of a feature of type VmbInt64

	Type	Name	Description
in	`const VmbInt64_t&`	value	The feature's value

5.12.15 SetValue()

Sets the value of a feature of type double

	Type	Name	Description
in	`const double&`	value	The feature's value

5.12.16 SetValue()

Sets the value of a feature of type char*

	Type	Name	Description
in	`const char*`	pValue	The feature's value

5.12.17 SetValue()

Sets the value of a feature of type bool

	Type	Name	Description
in	`bool`	value	The feature's value

5.12.18 SetValue()

Sets the value of a feature of type UcharVector

	Type	Name	Description
in	`const UcharVector&`	value	The feature's value

5.12.19 HasIncrement()

Gets the support state increment of a feature

	Type	Name	Description
out	`VmbBool_t&`	incrementsupported	The feature's increment support state

5.12.20 GetIncrement()

Gets the increment of a feature of type VmbInt64

	Type	Name	Description
out	`VmbInt64_t&`	increment	The feature's increment

5.12.21 GetIncrement()

Gets the increment of a feature of type double

	Type	Name	Description
out	`double&`	increment	The feature's increment

5.12.22 IsValueAvailable()

Indicates whether an existing enumeration value is currently available. An enumeration value might not be selectable due to the camera's current configuration.

	Type	Name	Description
in	`const char*`	pValue	The enumeration value as string
out	`bool&`	available	True when the given value is available

- **VmbErrorSuccess:** If no error
- **VmbErrorInvalidValue:** If the given value is not a valid enumeration value for this enum
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode
- **VmbErrorWrongType:** The feature is not an enumeration

5.12.23 IsValueAvailable()

Indicates whether an existing enumeration value is currently available. An enumeration value might not be selectable due to the camera's current configuration.

	Type	Name	Description
in	`const VmbInt64_t`	value	The enumeration value as int
out	`bool&`	available	True when the given value is available

- **VmbErrorSuccess:** If no error
- **VmbErrorInvalidValue:** If the given value is not a valid enumeration value for this enum
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode
- **VmbErrorWrongType:** The feature is not an enumeration

5.12.24 RunCommand()

Executes a feature of type Command

5.12.25 IsCommandDone()

Indicates whether the execution of a feature of type Command has finished

	Type	Name	Description
out	`bool&`	isDone	True when execution has finished

5.12.26 GetName()

Queries a feature's name

	Type	Name	Description
out	`std::string&`	name	The feature's name

5.12.27 GetDisplayName()

Queries a feature's display name

	Type	Name	Description
out	`std::string&`	displayName	The feature's display name

5.12.28 GetDataType()

Queries a feature's type

	Type	Name	Description
out	`VmbFeatureDataType&`	dataType	The feature's type

5.12.29 GetFlags()

Queries a feature's access status

	Type	Name	Description
out	`VmbFeatureFlagsType&`	flags	The feature's access status

5.12.30 GetCategory()

Queries a feature's category in the feature tress

	Type	Name	Description
out	`std::string&`	category	The feature's position in the feature tree

5.12.31 GetPollingTime()

Queries a feature's polling time

	Type	Name	Description
out	`VmbUint32_t&`	pollingTime	The interval to poll the feature

5.12.32 GetUnit()

Queries a feature's unit

	Type	Name	Description
out	`std::string&`	unit	The feature's unit

5.12.33 GetRepresentation()

Queries a feature's representation

	Type	Name	Description
out	`std::string&`	representation	The feature's representation

5.12.34 GetVisibility()

Queries a feature's visibility

	Type	Name	Description
out	`VmbFeatureVisibilityType&`	visibility	The feature's visibility

5.12.35 GetToolTip()

Queries a feature's tool tip to display in the GUI

	Type	Name	Description
out	`std::string&`	toolTip	The feature's tool tip

5.12.36 GetDescription()

Queries a feature's description

	Type	Name	Description
out	`std::string&`	description	The feature'sdescription

5.12.37 GetSFNCNamespace()

Queries a feature's Standard Feature Naming Convention namespace

	Type	Name	Description
out	`std::string&`	sFNCNamespace	The feature's SFNC namespace

5.12.38 GetAffectedFeatures()

Queries the feature's that are dependent from the current feature

	Type	Name	Description
out	`FeaturePtrVector&`	affectedFeatures	The features that get invalidated through the current feature

5.12.39 GetSelectedFeatures()

Gets the features that get selected by the current feature

	Type	Name	Description
out	`FeaturePtrVector&`	selectedFeatures	The selected features

5.12.40 IsReadable()

Queries the read access status of a feature

	Type	Name	Description
out	`bool&`	isReadable	True when feature can be read

5.12.41 IsWritable()

Queries the write access status of a feature

	Type	Name	Description
out	`bool&`	isWritable	True when feature can be written

5.12.42 IsStreamable()

Queries whether a feature's value can be transferred as a stream

	Type	Name	Description
out	`bool&`	isStreamable	True when streamable

5.12.43 RegisterObserver()

Registers an observer that notifies the application whenever a features value changes

	Type	Name	Description
out	`const IFeatureObserverPtr&`	pObserver	The observer to be registered

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pObserver" is NULL.

5.12.44 UnregisterObserver()

Unregisters an observer

	Type	Name	Description
out	`const IFeatureObserverPtr&`	pObserver	The observer to be unregistered

- **VmbErrorSuccess:** If no error
- **VmbErrorBadParameter:** "pObserver" is NULL.

5.13 EnumEntry

5.13.1 EnumEntry constructor

Creates an instance of class EnumEntry

	Type	Name	Description
in	`const char*`	pName	The name of the enum
in	`const char*`	pDisplayName	The declarative name of the enum
in	`const char*`	pDescription	The description of the enum
in	`const char*`	pTooltip	A tooltip that can be used by a GUI
in	`const char*`	pSNFCNamespace	The SFNC namespace of the enum
in	`VmbFeatureVisibility_t`	visibility	The visibility of the enum
in	`VmbInt64_t`	value	The integer value of the enum

5.13.2 EnumEntry constructor

Creates an instance of class EnumEntry

5.13.3 Copy constructor

Creates a copy of class EnumEntry

5.13.4 assignment operator

assigns EnumEntry to existing instance

5.13.5 EnumEntry destructor

Destroys an instance of class EnumEntry

5.13.6 GetName()

Gets the name of an enumeration

	Type	Name	Description
out	`std::string&`	name	The name of the enumeration

5.13.7 GetDisplayName()

Gets a more declarative name of an enumeration

	Type	Name	Description
out	`std::string&`	displayName	The display name of the enumeration

5.13.8 GetDescription()

Gets the description of an enumeration

	Type	Name	Description
out	`std::string&`	description	The description of the enumeration

5.13.9 GetTooltip()

Gets a tooltip that can be used as pop up help in a GUI

	Type	Name	Description
out	`std::string&`	tooltip	The tooltip as string

5.13.10 GetValue()

Gets the integer value of an enumeration

	Type	Name	Description
out	`VmbInt64_t&`	value	The integer value of the enumeration

5.13.11 GetVisibility()

Gets the visibility of an enumeration

	Type	Name	Description
out	`VmbFeatureVisibilityType&`	value	The visibility of the enumeration

5.13.12 GetSNFCNamespace()

Gets the standard feature naming convention namespace of the enumeration

	Type	Name	Description
out	`std::string&`	sFNCNamespace	The feature's SFNC namespace

5.14 AncillaryData

5.14.1 Open()

Opens the ancillary data to allow access to the elements of the ancillary data via feature access.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command

Note

This function can only succeed if the given frame has been filled by the API.

5.14.2 Close()

Closes the ancillary data inside a frame.

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid

Note

After reading the ancillary data and before re-queuing the frame, ancillary data must be closed.

5.14.3 GetBuffer()

Returns the underlying buffer

	Type	Name	Description
out	`VmbUchar_t*&`	pBuffer	A pointer to the buffer

- **VmbErrorSuccess:** If no error

5.14.4 GetBuffer()

Returns the underlying buffer

	Type	Name	Description
out	`const VmbUchar_t*&`	pBuffer	A pointer to the buffer

- **VmbErrorSuccess:** If no error

5.14.5 GetSize()

Returns the size of the underlying buffer

	Type	Name	Description
out	`VmbUint32_t&`	size	The size of the buffer

- **VmbErrorSuccess:** If no error