# Vimba C

# Manual

V1.4

2015-11-10

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

# Legal Notice

## Trademarks

## Warranty

## Copyright

### Allied Vision Technologies GmbH 11/2015

All rights reserved.
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 | 2012-11-15 | Initial version |
| 1.1 | 2013-02-22 | Different links, small changes |
| 1.2 | 2013-06-18 | Small corrections, layout changes |
| 1.3 | 2014-07-10 | Added function reference, re-structured and improved texts |
| 1.4 | 2015-11-10 | 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.

**www**

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 purpose of Vimba's APIs is to enable programmers to interact with Allied Vision cameras independent of the interface technology (1394, Gigabit Ethernet, USB). To achieve this, Vimba API utilizes GenICam transport layer modules to connect to the various camera interfaces and is therefore generic in terms of camera interfaces.

For accessing functionality of either Vimba or the connected cameras, you have two ways of control: the fixed set of API functions on the one hand and on the other hand using GenICam Features by calling functions like e.g. VmbFeatureXXXSet or VmbFeatureXXXGet on entities like Vimba or the cameras.

This manual mainly deals with the API functions.

**Note**

The Vimba Manual contains a description of the API concepts.

**Note**

Features are listed in the following documents:

- 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)

**www**

The latest GigE and USB Features Manuals are available from: http://www.alliedvision.com/en/support/technical-documentation.html

# 4 API usage

## 4.1 API Version

Even if new features are introduced to Vimba C API, your software remains backward compatible. Use `VmbVersionQuery` to check the version number of Vimba C API.

## 4.2 API Startup and Shutdown

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

- `VmbStartup` initializes Vimba API.
- `VmbShutdown` shuts down Vimba API (as soon as all callbacks are finished).

`VmbStartup` and `VmbShutdown` must always be paired. Calling the pair several times within the same program is possible, but not recommended. Only `VmbVersionQuery` can be run without initializing Vimba API. In order to free resources, shut down Vimba API when you don't use it.

## 4.3 Listing available cameras

**Note**

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

`VmbCamerasList` enumerates all cameras recognized by the underlying transport layers. With this command, the programmer can fetch all static details of a camera such as:

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

**GigE cameras:**
Listing cameras over the network is a two-step process:

1. To enable camera discovery events, run one of the following commands:
   - *GeVDiscoveryAllOnce* discovers all connected cameras once.
   - *GeVDiscoveryAllAuto* continually emits discovery packets and thus constantly consumes bandwidth. Use it only if you need to stay aware of changes to your network structure and new cameras.

   Both commands require a certain amount of time (*GeVDiscoveryAllDuration*) before returning.
2. To stop the camera discovery, run command *GeVDiscoveryAllOff*.

**USB and 1394 cameras:**
Changes to the plugged cameras are detected automatically. Consequently, any changes to the camera list are announced via discovery event.

All listed commands are applied to all network interfaces, see the example Listing 1.

Listing 1: Get Cameras

```
bool bGigE;
VmbUint32_t nCount;
VmbCameraInfo_t *pCameras;

// We ask Vimba for the presence of a GigE transport layer
VmbError_t err = VmbFeatureBoolGet( gVimbaHandle, "GeVTLIsPresent", &bGigE );
if ( VmbErrorSuccess == err )
{
    if ( true == bGigE )
    {
        // We use all network interfaces using the global Vimba handle
        err = VmbFeatureCommandRun( gVimbaHandle, "GeVDiscoveryAllOnce" );
    }
}
if ( VmbErrorSuccess == err )
{   // Get the number of connected cameras
    err = VmbCamerasList( NULL, 0, &nCount, sizeof *pCameras );

    if ( VmbErrorSuccess == err )
    {
        // Allocate accordingly
        pCameras = new VmbCameraInfo_t[ nCount ];
        // Get the cameras
        err = VmbCamerasList( pCameras, nCount, &nCount, sizeof *pCameras );
        // Print out each camera's name
        for ( VmbUint32_t i=0; i<nCount; ++i )
        {
            printf( " %s\n", pCameras[i].cameraName );
        }
    }
}
```

Struct VmbCameraInfo_t provides the entries listed in Table 1 for obtaining information about a camera.

| Struct Entry | Purpose |
|---|---|
| const char* cameraIdString | The unique ID |
| const char* cameraName | The name |
| const char* modelName | The model name |
| const char* serialString | The serial number |
| VmbAccessMode_t permittedAccess | The mode to open the camera |
| const char* interfaceIdString | The ID of the interface the camera is connected to |

Table 1: Struct VmbCameraInfo_t

## Enable notifications of changed camera states

To **get notified whenever a camera is detected, disconnected, or changes its open state**:

- Run command feature *GeVDiscoveryAllAuto* on the System entity (GigE cameras only).
- Use VmbFeatureInvalidationRegister to register a callback with the Vimba System that gets executed on the according event. The function pointer to the callback function has to be of type VmbInvalidationCallback*.

**Note**

VmbShutdown blocks until all callbacks have finished execution.

**Caution**

Functions that must **not** be called within the camera notification callback:

- VmbStartup
- VmbShutdown
- VmbFeatureIntSet (and any other VmbFeature*Set function)
- VmbFeatureCommandRun

## 4.4 Opening and closing a camera

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

Call `VmbCameraOpen` and provide the ID of the camera as well as the desired access mode.

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.

When a camera has been opened successfully, a handle for further access is returned.

An example for **opening a camera** retrieved from the camera list is shown in Listing 2.

Listing 2: Open Camera

```
VmbCameraInfo_t *pCameras;
VmbHandle_t hCamera;

// Get all known cameras as described in chapter "Listing available cameras"

// Open the first camera
if ( VmbErrorSuccess == VmbCameraOpen(  pCameras[0].cameraIdString,
                                        VmbAccessModeFull, hCamera ) )
{
    printf( "Camera opened, handle [0x%p] retrieved.\n", hCamera );
}
```

Listing 3 shows how to **close a camera** using `VmbCameraClose` and the previously retrieved handle.

Listing 3: Close Camera

```
if ( VmbErrorSuccess == VmbCameraClose( hCamera ) )
{
    printf( "Camera closed.\n" );
}
```

## 4.5 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 camera and Vimba 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)

Vimba API provides several feature types, which all have their specific properties and functionalities, as shown in Table 2.

As shown in Table 2, Vimba API provides its own set of access functions for every feature data type. The static properties of a feature are held in struct `VmbFeatureInfo_t` as listed in Table 3. It may be filled by calling `VmbFeatureInfoQuery` for an individual feature, or by calling `VmbFeaturesList` for the whole list of features. Since not all features are available all the time, it is necessary to query their current accessibility by calling function `VmbFeatureAccessQuery`.

To **query all available features** of a camera, use `VmbFeaturesList`. This list does not change while the camera is opened as shown in Listing 4.

Listing 4: Get Features

```
VmbFeatureInfo_t *pFeatures;
VmbUint32_t nCount = 0;
VmbHandle_t hCamera;

// Open the camera as shown in chapter "Opening a camera"

// Get the number of features
VmbError_t err = VmbFeaturesList( hCamera, NULL, 0, &nCount, sizeof *pFeatures );

if ( VmbErrorSuccess == err && 0 < nCount )
{
    // Allocate accordingly
    pFeatures = new VmbFeatureInfo_t[ nCount ];

    // Get the features
    err = VmbFeaturesList(  hCamera, pFeatures, nCount, &nCount,
                            sizeof *pFeatures );

    // Print out their name and data type
    for ( int i=0; i<nCount; ++i )
    {
        printf("Feature '%s' of type: %d\n", pFeatures[i].name,
                                             pFeatures[i].featureDataType);
    }
}
```

| Feature Type | Operation | Function |
|---|---|---|
| Enumeration | Set | VmbFeatureEnumSet |
| | Get | VmbFeatureEnumGet |
| | Range | VmbFeatureEnumRangeQuery |
| | Other | VmbFeatureEnumIsAvailable<br>VmbFeatureEnumAsInt<br>VmbFeatureEnumAsString<br>VmbFeatureEnumEntryGet |
| Integer | Set | VmbFeatureIntSet |
| | Get | VmbFeatureIntGet |
| | Range | VmbFeatureIntRangeQuery |
| | Other | VmbFeatureIntIncrementQuery |
| Float | Set | VmbFeatureFloatSet |
| | Get | VmbFeatureFloatGet |
| String | Set | VmbFeatureStringSet |
| | Get | VmbFeatureStringGet |
| | Range | VmbFeatureStringMaxlengthQuery |
| Boolean | Set | VmbFeatureBoolSet |
| | Get | VmbFeatureBoolGet |
| Command | Set | VmbFeatureCommandRun |
| | Get | VmbFeatureCommandIsDone |
| Raw data | Set | VmbFeatureRawSet |
| | Get | VmbFeatureRawGet |
| | Range | VmbFeatureRawLengthQuery |

Table 2: Feature types and functions for reading and writing them

For an example of **reading a camera feature**, see Listing 5.

Listing 5: Reading a camera feature

```
VmbHandle_t hCamera;

// Open the camera as shown in chapter "Opening a camera"

VmbInt64_t nWidth;

if ( VmbErrorSuccess == VmbFeatureIntGet(   hCamera, "Width", &nWidth ))
{
    printf("Width: %ld\n", nPayloadSize);
}
```

As an example for **writing features to a camera** and **running a command feature**, see Listing 6.

Listing 6: Writing features and running command features

```
VmbHandle_t hCamera;
```

| Struct entry | Purpose |
|---|---|
| `const char* name` | Name used in the API |
| `VmbFeatureData_t featureDataType` | Data type of this feature |
| `VmbFeatureFlags_t featureFlags` | Access flags for this feature |
| `const char* category` | Category this feature can be found in |
| `const char* displayName` | Feature name to be used in GUIs |
| `VmbUint32_t pollingTime` | Predefined polling time for volatile features |
| `const char* unit` | Measuring unit as given in the XML file |
| `const char* representation` | Representation of a numeric feature |
| `VmbFeatureVisibility_t visibility` | GUI visibility |
| `const char* tooltip` | Short description, e.g. for a tooltip |
| `const char* description` | Longer description |
| `const char* sfncNamespace` | Namespace this feature resides in |
| `VmbBool_t isStreamable` | Indicates if a feature can be stored to or loaded from a file |
| `VmbBool_t hasAffectedFeatures` | Indicates if the feature potentially affects other features |
| `VmbBool_t hasSelectedFeatures` | Indicates if the feature selects other features |

Table 3: Struct `VmbFeatureInfo_t`

```

// Open the camera as shown in chapter "Opening a camera"

if ( VmbErrorSuccess == VmbFeatureEnumSet( hCamera, "AcquisitionMode",
                                           "Continuous" ))
{
    if ( VmbErrorSuccess = VmbFeatureCommandRun( hCamera, "AcquisitionStart" ))
    {
        printf("Acquisition successfully started\n");
    }
}
```

Table 4 introduces the **basic features of all cameras**. A feature has a name, a type, and access flags such as `read-permitted` and `write-permitted`.

To **get notified whenever a feature's value changes**, use `VmbFeatureInvalidationRegister` to register a callback that gets executed on the according event. For camera features, use the camera handle for registration. The function pointer to the callback function has to be of type `VmbInvalidationCallback*`.

**Note**

`VmbShutdown` only returns after all callbacks have finished execution.

| Feature | Type | Access Flags | Description |
|---|---|---|---|
| *AcquisitionMode* | Enumeration | R/W | The acquisition mode of the camera. Value set: 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 4: Basic features found on all cameras

**Caution**

Functions that must **not** be called within a feature invalidation callback:

- `VmbStartup`
- `VmbShutdown`
- `VmbFeatureIntSet` (and any other `VmbFeature*Set` function)
- `VmbFeatureCommandRun`

## 4.6 Image Capture (API) and Acquisition (Camera)

**Note**

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

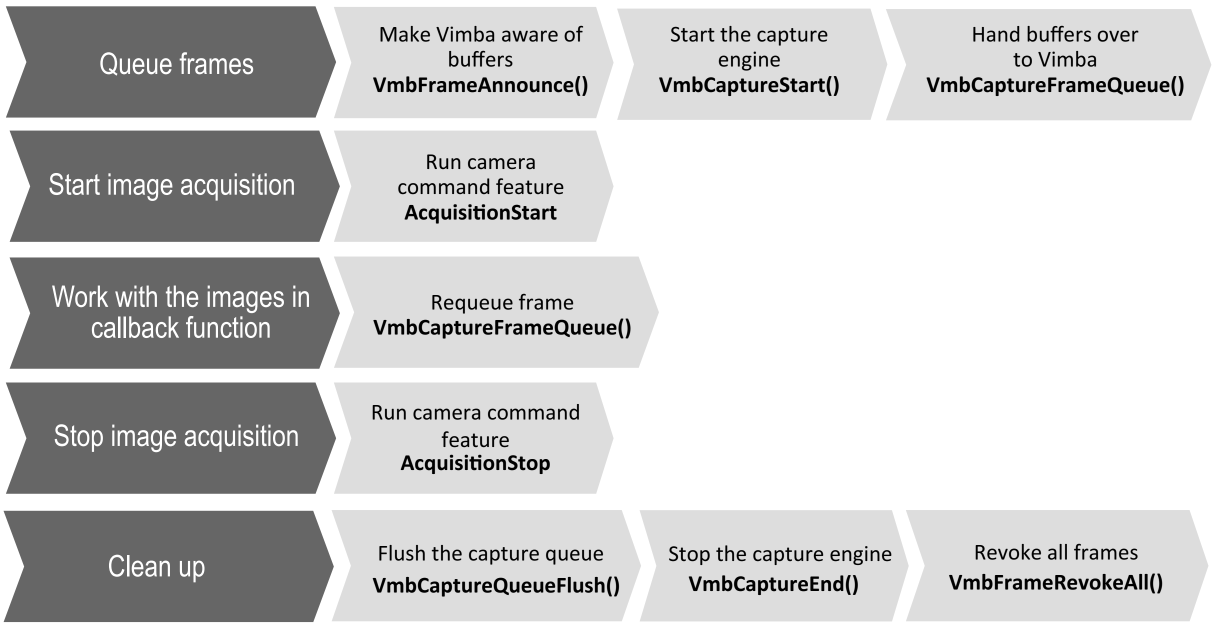
**Note**

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

### 4.6.1 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:



Queue frames
Make Vimba aware of buffers VmbFrameAnnounce()
Start the capture engine VmbCaptureStart()
Hand buffers over to Vimba VmbCaptureFrameQueue()
Start image acquisition
Run camera command feature AcquisitionStart
Work with the images in callback function
Requeue frame VmbCaptureFrameQueue()
Stop image acquisition
Run camera command feature AcquisitionStop
Clean up
Flush the capture queue VmbCaptureQueueFlush()
Stop the capture engine VmbCaptureEnd()
Revoke all frames VmbFrameRevokeAll()

Figure 1: Typical asynchronous application using Vimba C

### 4.6.2 Image Capture

**Note**

ⓘ The bracketed tokens in this chapter refer to Listing 7.

To enable image capture, frame buffers must be allocated and the API must be prepared for incoming frames:

To capture images sent by the camera, follow these steps:

1. Open the camera as described in chapter Opening and closing a camera
2. Query the necessary buffer size through the feature *PayloadSize* (A). Allocate frame buffers of this size (B).
3. Announce the frame buffers (1).
4. Start the capture engine (2).
5. Queue the frame you have just created with `VmbCaptureFrameQueue`, 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. How you proceed depends on the acquisition model you need:
   - **Synchronous:** Use `VmbCaptureFrameWait` to receive an image frame while blocking your execution thread.
   - **Asynchronous:** Register a callback (C) that gets executed when capturing is complete. Use the camera handle for registration. The function pointer to the callback function has to be of type `VmbFrameCallback*`. Within the callback routine, queue the frame again after you have processed it.
6. Call `VmbCaptureQueueFlush` to cancel all frames on the queue.
7. Stop the capture engine with `VmbCaptureEnd`.
8. Revoke the frames with `VmbFrameRevokeAll` to clear the buffers.

To assure correct continuous image capture, queue 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 `VmbFrame_t.receiveStatus` equals `VmbFrameStatusComplete` when a frame is returned to ensure the data is valid.

**Caution**

Functions that must **not** be called within the Frame callback routine.

- `VmbStartup`
- `VmbShutdown`
- `VmbCameraOpen`
- `VmbCameraClose`
- `VmbFrameAnnounce`
- `VmbFrameRevoke`
- `VmbFrameRevokeAll`
- `VmbCaptureStart`
- `VmbCaptureStop`

### 4.6.3 Image Acquisition

As soon as the API is prepared (see chapter Image Capture), you can 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 7 shows a **simplified streaming example** (without error handling).

Listing 7: Streaming

```c
#define FRAME_COUNT 3              // We choose to use 3 frames
VmbError_t err;                    // Vimba functions return an error code that the
                                   // programmer should check for VmbErrorSuccess
VmbHandle_t hCamera                // A handle to our opened camera
VmbFrame_t frames[FRAME_COUNT];    // A list of frames for streaming
VmbUInt64_t nPLS;                  // The payload size of one frame

// The callback that gets executed on every filled frame
void VMB_CALL FrameDoneCallback( const VmbHandle_t hCamera, VmbFrame_t *pFrame )
{
    if ( VmbFrameStatusComplete == pFrame->receiveStatus )
    {
        printf( "Frame successfully received\n" );
    }
    else
    {
        printf( "Error receiving frame\n" );
    }
    VmbCaptureFrameQueue( hCamera, pFrame, FrameDoneCallback );
}

// Get all known cameras as described in chapter "List available cameras"
// and open the camera as shown in chapter "Opening a camera"

// Get the required size for one image
err = VmbFeatureIntGet( hCamera, "PayloadSize", &nPLS );                    (A)
for ( int i=0; i<FRAME_COUNT; ++i )
{
    // Allocate accordingly
    frames[i].buffer = new char[ nPLS ];                                    (B)
    frames[i].bufferSize = nPLS;                                            (B)
    // Anounce the frame
    VmbFrameAnnounce( hCamera, frames[i], sizeof(VmbFrame_t) );             (1)
}

// Start capture engine on the host
err = VmbCaptureStart( hCamera );                                           (2)

// Queue frames and register callback
for ( int i=0; i<FRAME_COUNT; ++i )
{
    VmbCaptureFrameQueue(   hCamera, frames[i],                             (3)
                            FrameDoneCallback );                            (C)
}

// Start acquisition on the camera
err = VmbFeatureCommandRun( hCamera, "AcquisitionStart" );                  (4)
```

## 4.7 Using Events

Events serve many purposes and can have several origins, e.g. generic camera events or just feature changes.

All of these cases are handled in Vimba C uniformly with the same mechanism: You simply register a notification callback with `VmbFeatureInvalidationRegister` for the feature of your choice which gets called when there is a change to that feature.

Three examples are listed in this chapter:

- Camera list notifications
- Camera event features
- Tracking invalidations of features

See Listing 8 for an example of being notified about **camera list changes**. (For more details about System features see the Vimba Features Manual)

Listing 8: Getting notified about camera list changes

```
// 1. define callback function
void VMB_CALL CameraListCB(VmbHandle_t handle, const char* name, void* context)
{
    char cameraName[255];

    VmbFeatureStringGet( handle, "DiscoveryCameraIdent", cameraName);
    printf( "Event was fired by camera %s\n", cameraName );
}

// 2. register the callback for that event
VmbFeatureInvalidationRegister( gVimbaHandle, "DiscoveryCameraEvent",
                                CameraListCB, NULL);

// 3. for GigE cameras, invoke "GeVDiscoveryAllOnce"
VmbFeatureCommandRun( gVimbaHandle, "GeVDiscoveryAllOnce");
```

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

Listing 9: Getting notified about feature changes

```
// 1. define callback function
void VMB_CALL WidthChangeCB(VmbHandle_t handle, const char* name, void* context)
{
    printf( "Feature changed: %s\n", name );
}

// 2. register callback for changes to Width
VmbFeatureInvalidationRegister( cameraHandle, "Width", WidthChangeCB, NULL);

// as an example, binning is changed, so the callback will be run
VmbFeatureIntegerSet( cameraHandle, "Binning", 4);
```

**Camera events** are also handled with the same mechanism of feature invalidation. See Listing 10 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 10: Getting notified about camera events

```
// 1. define callback function
void VMB_CALL EventCB(VmbHandle_t handle, const char* name, void* context)
{
    printf( "Event was fired: %s\n", name );
}

// 2. select "AcquisitionStart" event
VmbFeatureStringSet( cameraHandle, "EventSelector", "AcquisitionStart");

// 3. switch on the event notification
VmbFeatureEnumSet (cameraHandle, "EventNotification", "On");

// 4. register the callback for that event
VmbFeatureInvalidationRegister( cameraHandle, "EventAcquisitionStart",
                                EventCB, NULL);
```

## 4.8 Additional configuration: Listing Interfaces

`VmbInterfacesList` enumerates all Interfaces (GigE, USB, or 1394 adapters) recognized by the underlying transport layers.
See Listing 11 for an example.

Listing 11: Get Interfaces

```
VmbUint32_t nCount;
VmbInterfaceInfo_t *pInterfaces;

// Get the number of connected interfaces
VmbInterfacesList( NULL, 0, &nCount, sizeof *pInterfaces );

// Allocate accordingly
pInterfaces = new VmbInterfaceInfo_t[ nCount ];

// Get the interfaces
VmbInterfacesList( pCameras, nCount, &nCount, sizeof *pInterfaces );
```

Struct `VmbInterfaceInfo_t` provides the information about an interface as listed in Table 5.

| Struct entry | Purpose |
|---|---|
| `const char* interfaceIdString` | The unique ID |
| `VmbInterface_t interfaceType` | The camera interface type |
| `const char* interfaceName` | The name |
| `const char* serialString` | The serial number (not in use) |
| `VmbAccessMode_t permittedAccess` | The mode to open the interface |

Table 5: Struct `VmbInterfaceInfo_t`

To **get notified whenever an interface is detected or disconnected**, use `VmbFeatureInvalidationRegister` to register a callback that gets executed on the according event. Use the global Vimba handle for registration. The function pointer to the callback function has to be of type `VmbInvalidationCallback*`.

**Note**

`VmbShutdown` blocks until all callbacks have finished execution.

**Caution**

The list of functions that must **not** be called within the callback routine:

- `VmbStartup`
- `VmbShutdown`
- `VmbFeatureIntSet` (and any other `VmbFeature*Set` function)
- `VmbFeatureCommandRun`

## 4.9 Troubleshooting

### 4.9.1 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.

### 4.9.2 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

## 4.10 Error Codes

All Vimba API functions return an error code of type `VmbErrorType`.
Typical errors are listed with each function in chapter Function reference. However, any of the error codes listed in Table 6 might be returned.

| Error Code | Value | Description |
|---|---|---|
| `VmbErrorSuccess` | 0 | No error |
| `VmbErrorInternalFault` | -1 | Unexpected fault in Vimba or driver |
| `VmbErrorApiNotStarted` | -2 | `VmbStartup` 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 6: Error codes returned by Vimba

# 5 Function reference

In this chapter, you can find a complete list of all methods that are described in `VimbaC.h`.

All function and type definitions are designed to be platform-independent and portable from other languages.

**General conventions:**

- Method names are composed in the following manner:
  - Vmb"Action". Example: `VmbStartup()`
  - Vmb"Entity""Action". Example: `VmbInterfaceOpen()`
  - Vmb"ActionTarget""Action". Example: `VmbFeaturesList()`
  - Vmb"Entity""SubEntity""Action". Example: `VmbFeatureCommandRun()`
- Methods dealing with features, memory, or registers accept a handle from the following entity list as first parameter: System, Camera, Interface, and AncillaryData. All other methods taking handles accept only a specific handle.
- Strings (generally declared as "const char *") are assumed to have a trailing 0 character.
- All pointer parameters should of course be valid, except if stated otherwise.
- To ensure compatibility with older programs linked against a former version of the API, all struct* parameters have an accompanying sizeofstruct parameter.
- Functions returning lists are usually called twice: once with a zero buffer to get the length of the list, and then again with a buffer of the correct length.

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 are listed
- Finally, a more detailed description about the function is given

## 5.1 Callbacks

### 5.1.1 VmbInvalidationCallback

Invalidation Callback type for a function that gets called in a separate thread and has been registered with VmbFeatureInvalidationRegister()

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **in** | `void*` | pUserContext | Pointer to the user context, see VmbFeatureInvalidationRegister |

**Note**

While the callback is run, all feature data is atomic. After the callback finishes, the feature data might be updated with new values.

**Caution**

Do not spend too much time in this thread; it will prevent the feature values from being updated from any other thread or the lower-level drivers.

### 5.1.2 VmbFrameCallback

Frame Callback type for a function that gets called in a separate thread if a frame has been queued with VmbCaptureFrameQueue()

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | cameraHandle | Handle of the camera |
| **out** | `VmbFrame_t*` | pFrame | Frame completed |

## 5.2 API Version

### 5.2.1 VmbVersionQuery()

Retrieve the version number of VimbaC.

| | Type | Name | Description |
|---|---|---|---|
| **out** | `VmbVersionInfo_t*` | pVersionInfo | Pointer to the struct where version information is copied |
| **in** | `VmbUint32_t` | sizeofVersionInfo | Size of structure in bytes |

- **VmbErrorSuccess:** If no error
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API
- **VmbErrorBadParameter:** "pVersionInfo" is NULL.

**Note**

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

## 5.3 API Initialization

### 5.3.1 VmbStartup()

Initialize the VimbaC API.

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

**Note**

On successful return, the API is initialized; this is a necessary call.

**Caution**

This method must be called before any VimbaC function other than VmbVersion-Query() is run.

### 5.3.2 VmbShutdown()

Perform a shutdown on the API.

**Note**

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

## 5.4 Camera Enumeration & Information

### 5.4.1 VmbCamerasList()

Retrieve a list of all cameras.

| | Type | Name | Description |
|---|---|---|---|
| **out** | `VmbCameraInfo_t*` | pCameraInfo | Array of VmbCameraInfo_t, allocated by the caller. The camera list is copied here. May be NULL if pNumFound is used for size query. |
| **in** | `VmbUint32_t` | listLength | Number of VmbCameraInfo_t elements provided |
| **out** | `VmbUint32_t*` | pNumFound | Number of VmbCameraInfo_t elements found. |
| **in** | `VmbUint32_t` | sizeofCameraInfo | Size of the structure (if pCameraInfo == NULL this parameter is ignored) |

- **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:** The given list length was insufficient to hold all available entries
- **VmbErrorBadParameter:** The pNumFound parameter was NULL

**Note**

Camera detection is started with the registration of the "DiscoveryCameraEvent" event or the first call of VmbCamerasList(), which may be delayed if no "DiscoveryCameraEvent" event is registered (see examples). VmbCamerasList() is usually called twice: once with an empty array to query the list length, and then again with an array of the correct length. If camera lists change between the calls, pNumFound may deviate from the query return.

### 5.4.2 VmbCameraInfoQuery()

Retrieve information on a camera given by an ID.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const char*` | idString | ID of the camera |
| **out** | `VmbCameraInfo_t*` | pInfo | Structure where information will be copied. May be NULL. |
| **in** | `VmbUint32_t` | sizeofCameraInfo | Size of the structure |

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorNotFound:** The designated camera cannot be found
- **VmbErrorStructSize:** The given struct size is not valid for this API version
- **VmbErrorBadParameter:** idString was NULL

**Note**

May be called if a camera has not been opened by the application yet. Examples for "idString": "DEV_81237473991" for an ID given by a transport layer, "169.254.12.13" for an IP address, "000F314C4BE5" for a MAC address or "1234567890" for a plain serial number.

### 5.4.3 VmbCameraOpen()

Open the specified camera.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const char*` | idString | ID of the camera |
| **in** | `VmbAccessMode_t` | accessMode | Determines the level of control you have on the camera |
| **out** | `VmbHandle_t*` | pCameraHandle | A camera handle |

- **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
- **VmbErrorInvalidCall:** if called from frame callback
- **VmbErrorBadParameter:** if idString or pCameraHandle is NULL

**Note**

A camera may be opened in a specific access mode, which determines the level of control you have on a camera. Examples for "idString": "DEV_81237473991" for an ID given by a transport layer, "169.254.12.13" for an IP address, "000F314C4BE5" for a MAC address or "1234567890" for a plain serial number.

### 5.4.4 VmbCameraClose()

Close the specified camera.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | cameraHandle | A valid camera handle |

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

**Note**

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

## 5.5 Features

### 5.5.1 VmbFeaturesList()

List all the features for this entity.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **out** | `VmbFeatureInfo_t*` | pFeatureInfoList | An array of VmbFeatureInfo_t to be filled by the API. May be NULL if pNumFund is used for size query. |
| **in** | `VmbUint32_t` | listLength | Number of VmbFeatureInfo_t elements provided |
| **out** | `VmbUint32_t*` | pNumFound | Number of VmbFeatureInfo_t elements found. May be NULL if pFeatureInfoList is not NULL. |
| **in** | `VmbUint32_t` | sizeofFeatureInfo | Size of a VmbFeatureInfo_t entry |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorStructSize:** The given struct size of VmbFeatureInfo_t is not valid for this version of the API
- **VmbErrorMoreData:** The given list length was insufficient to hold all available entries

**Note**

This method lists all implemented features, whether they are currently available or not. The list of features does not change as long as the camera/interface is connected. "pNumFound" returns the number of VmbFeatureInfo elements. This function is usually called twice: once with an empty list to query the length of the list, and then again with an list of the correct length.

### 5.5.2 VmbFeatureInfoQuery()

Query information about the constant properties of a feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **out** | `VmbFeatureInfo_t*` | pFeatureInfo | The feature info to query |
| **in** | `VmbUint32_t` | sizeofFeatureInfo | Size of the structure |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API

**Note**

Users provide a pointer to VmbFeatureInfo_t, which is then set to the internal representation.

### 5.5.3 VmbFeatureListAffected()

List all the features that might be affected by changes to this feature.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the feature |
| out | `VmbFeatureInfo_t*` | pFeatureInfoList | An array of VmbFeatureInfo_t to be filled by the API. May be NULL if pNumFound is used for size query. |
| in | `VmbUint32_t` | listLength | Number of VmbFeatureInfo_t elements provided |
| out | `VmbUint32_t*` | pNumFound | Number of VmbFeatureInfo_t elements found. May be NULL is pFeatureInfoList is not NULL. |
| in | `VmbUint32_t` | sizeofFeatureInfo | Size of a VmbFeatureInfo_t entry |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorStructSize:** The given struct size of VmbFeatureInfo_t is not valid for this version of the API
- **VmbErrorMoreData:** The given list length was insufficient to hold all available entries

**Note**

This method lists all affected features, whether they are currently available or not. The value of affected features depends directly or indirectly on this feature (including all selected features). The list of features does not change as long as the camera/interface is connected. This function is usually called twice: once with an empty array to query the length of the list, and then again with an array of the correct length.

### 5.5.4 VmbFeatureListSelected()

List all the features selected by a given feature for this module.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the feature |
| out | `VmbFeatureInfo_t*` | pFeatureInfoList | An array of VmbFeatureInfo_t to be filled by the API. May be NULL if pNumFound is used for size query. |
| in | `VmbUint32_t` | listLength | Number of VmbFeatureInfo_t elements provided |
| out | `VmbUint32_t*` | pNumFound | Number of VmbFeatureInfo_t elements found. May be NULL if pFeatureInfoList is not NULL. |
| in | `VmbUint32_t` | sizeofFeatureInfo | Size of a VmbFeatureInfo_t entry |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API
- **VmbErrorMoreData:** The given list length was insufficient to hold all available entries

**Note**

This method lists all selected features, whether they are currently available or not. Features with selected features ("selectors") have no direct impact on the camera, but only influence the register address that selected features point to. The list of features does not change while the camera/interface is connected. This function is usually called twice: once with an empty array to query the length of the list, and then again with an array of the correct length.

### 5.5.5 VmbFeatureAccessQuery()

Return the dynamic read and write capabilities of this feature.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features. |
| in | `const char *` | name | Name of the feature. |
| out | `VmbBool_t *` | pIsReadable | Indicates if this feature is readable. May be NULL. |
| out | `VmbBool_t *` | pIsWriteable | Indicates if this feature is writable. May be NULL. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorBadParameter:** pIsReadable and pIsWriteable were both NULL
- **VmbErrorNotFound:** The feature was not found

**Note**

The access mode of a feature may change. For example, if "PacketSize" is locked while image data is streamed, it is only readable.

## 5.6 Integer

### 5.6.1 VmbFeatureIntGet()

Get the value of an integer feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **out** | `VmbInt64_t*` | pValue | Value to get |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Integer
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** If name or pValue is NULL

### 5.6.2 VmbFeatureIntSet()

Set the value of an integer feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **in** | `VmbInt64_t` | value | Value to set |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Integer
- **VmbErrorInvalidValue:** "value" is either out of bounds or not an increment of the minimum
- **VmbErrorBadParameter:** If name is NULL
- **VmbErrorNotFound:** If the feature was not found
- **VmbErrorInvalidCall:** If called from frame callback

### 5.6.3 VmbFeatureIntRangeQuery()

Query the range of an integer feature.

| | Type | Name | Description |
|---|---|---|---|
| in | const VmbHandle_t | handle | Handle for an entity that exposes features |
| in | const char* | name | Name of the feature |
| out | VmbInt64_t* | pMin | Minimum value to be returned. May be NULL. |
| out | VmbInt64_t* | pMax | Maximum value to be returned. May be NULL. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorBadParameter:** If name is NULL or pMin and pMax are NULL
- **VmbErrorWrongType:** The type of feature "name" is not Integer
- **VmbErrorNotFound:** If the feature was not found

### 5.6.4 VmbFeatureIntIncrementQuery()

Query the increment of an integer feature.

| | Type | Name | Description |
|---|---|---|---|
| in | const VmbHandle_t | handle | Handle for an entity that exposes features |
| in | const char* | name | Name of the feature |
| out | VmbInt64_t* | pValue | Value of the increment to get. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Integer
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter: IF name or pValue is NULL**

## 5.7 Float

### 5.7.1 VmbFeatureFloatGet()

Get the value of a float feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **out** | `double*` | pValue | Value to get |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Float
- **VmbErrorBadParameter:** If name or pValue is NULL
- **VmbErrorNotFound:** The feature was not found

### 5.7.2 VmbFeatureFloatSet()

Set the value of a float feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **in** | `double` | value | Value to set |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Float
- **VmbErrorInvalidValue:** "value" is not within valid bounds
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** If name is NULL
- **VmbErrorInvalidCall:** If called from frame callback

### 5.7.3 VmbFeatureFloatRangeQuery()

Query the range of a float feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **out** | `double*` | pMin | Minimum value to be returned. May be NULL. |
| **out** | `double*` | pMax | Maximum value to be returned. May be NULL. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Float
- **VmbErrorNotFound:** The feature was not found
- **VmbBadParameter:** If name is NULL or pMin and pMax are NULL

**Note**

Only one of the values may be queried if the other parameter is set to NULL, but if both parameters are NULL, an error is returned.

### 5.7.4 VmbFeatureFloatIncrementQuery()

Query the increment of an float feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **out** | `double*` | pValue | Value of the increment to get. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Integer
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter: IF name or pValue is NULL**

## 5.8 Enum

### 5.8.1 VmbFeatureEnumGet()

Get the value of an enumeration feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **out** | `const char**` | pValue | The current enumeration value. The returned value is a reference to the API value |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** if name or pValue is NULL

### 5.8.2 VmbFeatureEnumSet()

Set the value of an enumeration feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **in** | `const char*` | value | Value to set |

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode
- **VmbErrorBadHandle:** The given handle is not valid
- **VmbErrorInvalidAccess:** Operation is invalid with the current access mode
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorInvalidValue:** "value" is not within valid bounds
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** If name ore value is NULL
- **VmbErrorInvalidCall:** If called from frame callback

### 5.8.3 VmbFeatureEnumRangeQuery()

Query the value range of an enumeration feature.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the feature |
| out | `const char*` | const* | pNameArray An Array of enumeration value names may be NULL if pNumFilled is used for size query |
| in | `VmbUint32_t` | arrayLength | Number of elements in the array |
| out | `VmbUint32_t *` | pNumFilled | Number of filled elements my be NULL if pNameArray is not NULL |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorMoreData:** The given array length was insufficient to hold all available entries
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** if name is NULL or pNameArray and pNumFilled are NULL

### 5.8.4 VmbFeatureEnumIsAvailable()

Check if a certain value of an enumeration is available.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the feature |
| in | `const char*` | value | Value to check |
| out | `VmbBool_t *` | pIsAvailable | Indicates if the given enumeration value is available |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** if name or value or pIsAvailable is NULL

### 5.8.5 VmbFeatureEnumAsInt()

Get the integer value for a given enumeration string value.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the feature |
| in | `const char*` | value | The enumeration value to get the integer value for |
| out | `VmbInt64_t*` | pIntVal | The integer value for this enumeration entry |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** if name or value pIntVal is NULL

**Note**

Converts a name of an enum member into an int value ("Mono12Packed" to 0x10C0006)

## 5.8.6 VmbFeatureEnumAsString()

Get the enumeration string value for a given integer value.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the feature |
| **in** | `VmbInt64_t` | intValue | The numeric value |
| **out** | `const char**` | pStringValue | The string value for the numeric value |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** If name or pStringValue is NULL

**Note**

Converts an int value to a name of an enum member (e.g. 0x10C0006 to "Mono12Packed")

## 5.8.7 VmbFeatureEnumEntryGet()

Get infos about an entry of an enumeration feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | featureName | Name of the feature |
| **in** | `const char*` | entryName | Name of the enum entry of that feature |
| **out** | `VmbFeatureEnumEntry_t*` | pFeatureEnumEntry | Infos about that entry returned by the API |
| **in** | `VmbUint32_t` | sizeofFeatureEnumEntry | Size of the structure |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorStructSize Size of VmbFeatureEnumEntry_t is not compatible with the API version**
- **VmbErrorWrongType:** The type of feature "name" is not Enumeration
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorBadParameter:** if featureName or entryName or pFeatureEnumEntry is NULL

## 5.9 String

### 5.9.1 VmbFeatureStringGet()

Get the value of a string feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the string feature |
| **out** | `char*` | buffer | String buffer to fill. May be NULL if pSizeFilled is used for size query. |
| **in** | `VmbUint32_t` | bufferSize | Size of the input buffer |
| **out** | `VmbUint32_t*` | pSizeFilled | Size actually filled my be NULL if buffer is not NULL. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorMoreData:** The given buffer size was too small
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorWrongType:** The type of feature "name" is not String

**Note**

This function is usually called twice: once with an empty buffer to query the length of the string, and then again with a buffer of the correct length.

### 5.9.2 VmbFeatureStringSet()

Set the value of a string feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the string feature |
| **in** | `const char*` | value | Value to set |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorNotFound:** The feature was not found
- **VmbErrorWrongType:** The type of feature "name" is not String
- **VmbErrorInvalidValue:** Length of "value" exceeded the maximum length
- **VmbErrorBadParameter:** if name or value is NULL
- **VmbErrorInvalidCall:** If called from frame callback

### 5.9.3 VmbFeatureStringMaxlengthQuery()

Get the maximum length of a string feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the string feature |
| **out** | `VmbUint32_t*` | pMaxLength | Maximum length of this string feature |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not String
- **VmbErrorBadParameter:** If name or pMaxLength is NULL

## 5.10 Boolean

### 5.10.1 VmbFeatureBoolGet()

Get the value of a boolean feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the boolean feature |
| **out** | `VmbBool_t *` | pValue | Value to be read |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Boolean
- **VmbErrorNotFound:** If feature is not found
- **VmbErrorBadParameter:** If name or pValue is NULL

### 5.10.2 VmbFeatureBoolSet()

Set the value of a boolean feature.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the boolean feature |
| **in** | `VmbBool_t` | value | Value to write |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Boolean
- **VmbErrorInvalidValue:** "value" is not within valid bounds
- **VmbErrorNotFound:** If the feature is not found
- **VmbErrorBadParameter:** name is NULL
- **VmbErrorInvalidCall:** If called from frame callback

## 5.11 Command

### 5.11.1 VmbFeatureCommandRun()

Run a feature command.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the command feature |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Command
- **VmbErrorNotFound:** Feature was not found
- **VmbErrorBadParameter:** name is NULL

### 5.11.2 VmbFeatureCommandIsDone()

Check if a feature command is done.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the command feature |
| **out** | `VmbBool_t *` | pIsDone | State of the command. |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Command
- **VmbErrorNotFound:** Feature was not found
- **VmbErrorBadParameter:** name or pIsDone is NULL

## 5.12 Raw

### 5.12.1 VmbFeatureRawGet()

Read the memory contents of an area given by a feature name.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the raw feature |
| out | `char*` | pBuffer | Buffer to fill |
| in | `VmbUint32_t` | bufferSize | Size of the buffer to be filled |
| out | `VmbUint32_t*` | pSizeFilled | Number of bytes actually filled |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Register
- **VmbErrorNotFound:** Feature was not found
- **VmbErrorBadParameter:** If name or pBuffer or pSizeFilled is NULL

**Note**

This feature type corresponds to a top-level "Register" feature in GenICam. Data transfer is split up by the transport layer if the feature length is too large. You can get the size of the memory area addressed by the feature "name" by VmbFeatureRawLengthQuery().

### 5.12.2 VmbFeatureRawSet()

Write to a memory area given by a feature name.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| in | `const char*` | name | Name of the raw feature |
| in | `const char*` | pBuffer | Data buffer to use |
| in | `VmbUint32_t` | bufferSize | Size of the buffer |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Register
- **VmbErrorNotFound:** Feature was not found
- **VmbErrorBadParameter:** If name or pBuffer is NULL
- **VmbErrorInvalidCall:** If called from frame callback

**Note**

This feature type corresponds to a first-level "Register" node in the XML file. Data transfer is split up by the transport layer if the feature length is too large. You can get the size of the memory area addressed by the feature "name" by VmbFeatureRawLengthQuery().

### 5.12.3 VmbFeatureRawLengthQuery()

Get the length of a raw feature for memory transfers.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that exposes features |
| **in** | `const char*` | name | Name of the raw feature |
| **out** | `VmbUint32_t*` | pLength | Length of the raw feature area (in bytes) |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorWrongType:** The type of feature "name" is not Register
- **VmbErrorNotFound:** Feature not found
- **VmbErrorBadParameter:** If name or pLength is NULL

**Note**

This feature type corresponds to a first-level "Register" node in the XML file.

## 5.13 Feature invalidation

### 5.13.1 VmbFeatureInvalidationRegister()

Register a VmbInvalidationCallback callback for feature invalidation signaling.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that emits events |
| in | `const char*` | name | Name of the event (NULL to register for any feature) |
| in | `VmbInvalidationCallback` | callback | Callback to be run, when invalidation occurs |
| in | `void*` | pUserContext | User context passed to function |

- **VmbErrorSuccess:** If no error
- **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

**Note**

Any feature change, either of its value or of its access state, may be tracked by registering an invalidation callback. Registering multiple callbacks for one feature invalidation event is possible because only the combination of handle, name, and callback is used as key. If the same combination of handle, name, and callback is registered a second time, it overwrites the previous one.

### 5.13.2 VmbFeatureInvalidationUnregister()

Unregister a previously registered feature invalidation callback.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | handle | Handle for an entity that emits events |
| in | `const char*` | name | Name of the event |
| in | `VmbInvalidationCallback` | callback | Callback to be removed |

- **VmbErrorSuccess:** If no error
- **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

**Note**

Since multiple callbacks may be registered for a feature invalidation event, a combination of handle, name, and callback is needed for unregistering, too.

## 5.14 Image preparation and acquisition

### 5.14.1 VmbFrameAnnounce()

Announce frames to the API that may be queued for frame capturing later.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle for a camera |
| in | `const VmbFrame_t*` | pFrame | Frame buffer to announce |
| in | `VmbUint32_t` | sizeofFrame | Size of the frame structure |

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given camera handle is not valid
- **VmbErrorBadParameter:** The given frame pointer is not valid or sizeofFrame is 0
- **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 into consideration by the API.

### 5.14.2 VmbFrameRevoke()

Revoke a frame from the API.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle for a camera |
| in | `const VmbFrame_t*` | pFrame | Frame buffer 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 camera handle is not valid
- **VmbErrorBadParameter:** The given frame pointer is not valid
- **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.

### 5.14.3 VmbFrameRevokeAll()

Revoke all frames assigned to a certain camera.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle for a camera |

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

### 5.14.4 VmbCaptureStart()

Prepare the API for incoming frames.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle for a 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

### 5.14.5 VmbCaptureEnd()

Stop the API from being able to receive frames.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle for a 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 input queue is flushed - The frame callback will not be called any more

### 5.14.6 VmbCaptureFrameQueue()

Queue frames that may be filled during frame capturing.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle of the camera |
| in | `const VmbFrame_t*` | pFrame | Pointer to an already announced frame |
| in | `VmbFrameCallback` | callback | Callback to be run when the frame is complete. NULL is Ok. |

- **VmbErrorSuccess:** If no error

- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given frame is not valid
- **VmbErrorStructSize:** The given struct size is not valid for this version of the API

**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 VmbFrameAnnounce() before, the application has to ensure that the frame is also revoked by calling VmbFrameRevoke() or VmbFrameRevokeAll() when cleaning up.

### 5.14.7 VmbCaptureFrameWait()

Wait for a queued frame to be filled (or dequeued).

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle of the camera |
| in | `const VmbFrame_t*` | pFrame | Pointer to an already announced & queued frame |
| in | `VmbUint32_t` | timeout | Timeout (in milliseconds) |

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

### 5.14.8 VmbCaptureQueueFlush()

Flush the capture queue.

| | Type | Name | Description |
|---|---|---|---|
| in | `const VmbHandle_t` | cameraHandle | Handle of the camera to flush |

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

**Note**

Control of 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.

## 5.15 Interface Enumeration & Information

### 5.15.1 VmbInterfacesList()

List all the interfaces currently visible to VimbaC.

| | Type | Name | Description |
|---|---|---|---|
| **out** | `VmbInterfaceInfo_t*` | pInterfaceInfo | Array of VmbInterfaceInfo_t, allocated by the caller. The interface list is copied here. May be NULL. |
| **in** | `VmbUint32_t` | listLength | Number of entries in the caller's pList array |
| **out** | `VmbUint32_t*` | pNumFound | Number of interfaces found (may be more than listLength!) returned here. |
| **in** | `VmbUint32_t` | sizeofInterfaceInfo | Size of one VmbInterfaceInfo_t entry |

- **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:** The given list length was insufficient to hold all available entries
- **VmbErrorBadParameter:** pNumFound was NULL

**Note**

All the interfaces known via GenICam TransportLayers are listed by this command and filled into the provided array. Interfaces may correspond to adapter cards or frame grabber cards or, in the case of FireWire to the whole 1394 infrastructure, for instance. This function is usually called twice: once with an empty array to query the length of the list, and then again with an array of the correct length.

### 5.15.2 VmbInterfaceOpen()

Open an interface handle for feature access.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const char*` | idString | The ID of the interface to get the handle for (returned by VmbInterfacesList()) |
| **out** | `VmbHandle_t*` | pInterfaceHandle | The handle for this interface. |

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

**Note**

An interface can be opened if interface-specific control or information is required, e.g. the number of devices attached to a specific interface. Access is then possible via feature access methods.

### 5.15.3 VmbInterfaceClose()

Close an interface.

| Type | | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | interfaceHandle | The handle of the interface to close. |

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

**Note**

After configuration of the interface, close it by calling this function.

## 5.16 Ancillary data

### 5.16.1 VmbAncillaryDataOpen()

Get a working handle to allow access to the elements of the ancillary data via feature access.

| | Type | Name | Description |
|---|---|---|---|
| in | `VmbFrame_t*` | pFrame | Pointer to a filled frame |
| out | `VmbHandle_t*` | pAncillaryDataHandle | Handle to the ancillary data inside the frame |

- **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.

### 5.16.2 VmbAncillaryDataClose()

Destroy the working handle to the ancillary data inside a frame.

| | Type | Name | Description |
|---|---|---|---|
| in | `VmbHandle_t` | ancillaryDataHandle | Handle to ancillary frame data |

- **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.

## 5.17 Memory/Register access

### 5.17.1 VmbMemoryRead()

Read an array of bytes.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that allows memory access |
| **in** | `VmbUint64_t` | address | Address to be used for this read operation |
| **in** | `VmbUint32_t` | bufferSize | Size of the data buffer to read |
| **out** | `char*` | dataBuffer | Buffer to be filled |
| **out** | `VmbUint32_t*` | pSizeComplete | Size of the data actually read |

- **VmbErrorSuccess:** If no error
- **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

### 5.17.2 VmbMemoryWrite()

Write an array of bytes.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that allows memory access |
| **in** | `VmbUint64_t` | address | Address to be used for this read operation |
| **in** | `VmbUint32_t` | bufferSize | Size of the data buffer to write |
| **in** | `const char*` | dataBuffer | Data to write |
| **out** | `VmbUint32_t*` | pSizeComplete | Number of bytes successfully written; if an error occurs this is less than bufferSize |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorMoreData:** Not all data were written; see pSizeComplete value for the number of bytes written

### 5.17.3 VmbRegistersRead()

Read an array of registers.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that allows register access |
| **in** | `VmbUint32_t` | readCount | Number of registers to be read |
| **in** | `const VmbUint64_t*` | pAddressArray | Array of addresses to be used for this read operation |
| **out** | `VmbUint64_t*` | pDataArray | Array of registers to be used for this read operation |
| **out** | `VmbUint32_t*` | pNumCompleteReads | Number of reads completed |

- **VmbErrorSuccess:** If no error
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command
- **VmbErrorBadHandle:** The given handle is not valid
- **VmbErrorIncomplete:** Not all the requested reads could be completed

**Note**

Two arrays of data must be provided: an array of register addresses and one for corresponding values to be read. The registers are read consecutively until an error occurs or all registers are written successfully.

## 5.17.4 VmbRegistersWrite()

Write an array of registers.

| | Type | Name | Description |
|---|---|---|---|
| **in** | `const VmbHandle_t` | handle | Handle for an entity that allows register access |
| **in** | `VmbUint32_t` | writeCount | Number of registers to be written |
| **in** | `const VmbUint64_t*` | pAddressArray | Array of addresses to be used for this write operation |
| **in** | `const VmbUint64_t*` | pDataArray | Array of reads to be used for this write operation |
| **out** | `VmbUint32_t*` | pNumCompleteWrites | Number of writes completed |

- **VmbErrorSuccess:** If no error
- **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
- **VmbErrorIncomplete:** Not all the requested writes could be completed

**Note**

Two arrays of data must be provided: an array of register addresses and one with the corresponding values to be written to these addresses. The registers are written consecutively until an error occurs or all registers are written successfully.