Allied Vision

Vimba

Vimba

Vimba C Manual

1.9.0

Legal Notice

Trademarks

Unless stated otherwise, all trademarks appearing in this document 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.

All rights reserved.

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

Contents

1 Contacting Allied Vision 8

2 Document history and conventions 9
2.1 Document history . . . 10
2.2 Conventions used in this manual . . . 10
2.2.1 Styles . . . 11
2.2.2 Symbols . . . 11

3 General aspects of the API 12

4 API usage 13
4.1 API Version . . . 14
4.2 API Startup and Shutdown . . . 14
4.3 Listing available cameras . . . 15
4.4 Opening and closing a camera . . . 18
4.5 Accessing Features . . . 20
4.6 Image Capture (API) and Acquisition (Camera) . . . 27
4.6.1 Image Capture and Image Acquisition . . . 27
4.6.2 Image Capture . . . 28
4.6.3 Image Acquisition . . . 29
4.7 Using Events . . . 33
4.8 Saving and loading settings . . . 35
4.9 Triggering cameras . . . 36
4.9.1 External trigger . . . 36
4.9.2 Trigger over Ethernet – Action Commands . . . 37
4.10 Additional configuration: Listing Interfaces . . . 39
4.11 Troubleshooting . . . 40
4.11.1 GigE cameras . . . 40
4.11.2 USB cameras . . . 41
4.11.3 Goldeye CL cameras . . . 41
4.12 Error Codes . . . 42

5 Function reference 43
5.1 Callbacks . . . 45
5.1.1 VmbInvalidationCallback . . . 45
5.1.2 VmbFrameCallback . . . 45
5.2 API Version . . . 46
5.2.1 VmbVersionQuery() . . . 46
5.3 API Initialization . . . 47
5.3.1 VmbStartup() . . . 47

5.3.2 VmbShutdown() . . . 47
5.4 Camera Enumeration & Information . . . 48
5.4.1 VmbCamerasList() . . . 48
5.4.2 VmbCameraInfoQuery() . . . 48
5.4.3 VmbCameraOpen() . . . 49
5.4.4 VmbCameraClose() . . . 50
5.5 Features . . . 51
5.5.1 VmbFeaturesList() . . . 51
5.5.2 VmbFeatureInfoQuery() . . . 51
5.5.3 VmbFeatureListAffected() . . . 52
5.5.4 VmbFeatureListSelected() . . . 53
5.5.5 VmbFeatureAccessQuery() . . . 54
5.6 Integer . . . 55
5.6.1 VmbFeatureIntGet() . . . 55
5.6.2 VmbFeatureIntSet() . . . 55
5.6.3 VmbFeatureIntRangeQuery() . . . 56
5.6.4 VmbFeatureIntIncrementQuery() . . . 56
5.7 Float . . . 58
5.7.1 VmbFeatureFloatGet() . . . 58
5.7.2 VmbFeatureFloatSet() . . . 58
5.7.3 VmbFeatureFloatRangeQuery() . . . 59
5.7.4 VmbFeatureFloatIncrementQuery() . . . 59
5.8 Enum . . . 61
5.8.1 VmbFeatureEnumGet() . . . 61
5.8.2 VmbFeatureEnumSet() . . . 61
5.8.3 VmbFeatureEnumRangeQuery() . . . 62
5.8.4 VmbFeatureEnumIsAvailable() . . . 62
5.8.5 VmbFeatureEnumAsInt() . . . 63
5.8.6 VmbFeatureEnumAsString() . . . 64
5.8.7 VmbFeatureEnumEntryGet() . . . 64
5.9 String . . . 66
5.9.1 VmbFeatureStringGet() . . . 66
5.9.2 VmbFeatureStringSet() . . . 66
5.9.3 VmbFeatureStringMaxlengthQuery() . . . 67
5.10 Boolean . . . 68
5.10.1 VmbFeatureBoolGet() . . . 68
5.10.2 VmbFeatureBoolSet() . . . 68
5.11 Command . . . 70
5.11.1 VmbFeatureCommandRun() . . . 70
5.11.2 VmbFeatureCommandIsDone() . . . 70
5.12 Raw . . . 71

5.12.1 VmbFeatureRawGet() . . . 71
5.12.2 VmbFeatureRawSet() . . . 71
5.12.3 VmbFeatureRawLengthQuery() . . . 72
5.13 Feature invalidation . . . 74
5.13.1 VmbFeatureInvalidationRegister() . . . 74
5.13.2 VmbFeatureInvalidationUnregister() . . . 74
5.14 Image preparation and acquisition . . . 76
5.14.1 VmbFrameAnnounce() . . . 76
5.14.2 VmbFrameRevoke() . . . 76
5.14.3 VmbFrameRevokeAll() . . . 77
5.14.4 VmbCaptureStart() . . . 77
5.14.5 VmbCaptureEnd() . . . 78
5.14.6 VmbCaptureFrameQueue() . . . 78
5.14.7 VmbCaptureFrameWait() . . . 79
5.14.8 VmbCaptureQueueFlush() . . . 79
5.15 Interface Enumeration & Information . . . 80
5.15.1 VmbInterfacesList() . . . 80
5.15.2 VmbInterfaceOpen() . . . 80
5.15.3 VmbInterfaceClose() . . . 81
5.16 Ancillary data . . . 82
5.16.1 VmbAncillaryDataOpen() . . . 82
5.16.2 VmbAncillaryDataClose() . . . 82
5.17 Memory/Register access . . . 83
5.17.1 VmbMemoryRead() . . . 83
5.17.2 VmbMemoryWrite() . . . 83
5.17.3 VmbRegistersRead() . . . 84
5.17.4 VmbRegistersWrite() . . . 84
5.17.5 VmbCameraSettingsSave() . . . 85
5.17.6 VmbCameraSettingsLoad() . . . 86

List of Tables

1 Struct VmbCameraInfo_t . . . 16
2 Enum VmbAccessModeType is represented as VmbUint32_t through VmbAccessMode_t 18
3 Feature types and functions for reading and writing them . . . 21
4 Struct VmbFeatureInfo_t . . . 22
5 Enum VmbFeatureDataType is represented as VmbUint32_t through VmbFeatureData_t . . . 22
6 Enum VmbFeatureFlagsType is represented as VmbUint32_t through VmbFeatureFlags_t . . . 23
7 Enum VmbFeatureVisibilityType is represented as VmbUint32_t through VmbFeatureVisibility_t . . . 23
8 Struct VmbFeatureEnumEntry_t . . . 23
9 Basic features found on all cameras . . . 25
10 Struct VmbFrame_t . . . 31
11 Enum VmbFrameStatusType is represented as VmbInt32_t through VmbFrameStatus_t 32
12 Enum VmbFrameFlagsType is represented as VmbUint32_t through VmbFrameFlags_t 32
13 Struct VmbFeaturePersistSettings_t . . . 35
14 Struct VmbInterfaceInfo_t . . . 39
15 Enum VmbInterfaceType is represented as VmbUint32_t through VmbInterfaceInfo_t . . . 40
16 Error codes returned by Vimba . . . 42

Listings

1 Get Cameras . . . 16
2 Open Camera . . . 18
3 Close Camera . . . 19
4 Get Features . . . 24
5 Reading a camera feature . . . 24
6 Writing features and running command features . . . 25
7 Streaming . . . 30
8 Getting notified about camera list changes . . . 33
9 Getting notified about feature changes . . . 34
10 Getting notified about camera events . . . 34
11 External trigger . . . 36
12 Action Commands . . . 38
13 Get Interfaces . . . 39

1 Contacting Allied Vision

Contact information on our website

https://www.alliedvision.com/en/meta-header/contact-us

Find an Allied Vision office or distributor

https://www.alliedvision.com/en/about-us/where-we-are

Email

info@alliedvision.com
support@alliedvision.com

Sales Offices

EMEA: +49 36428-677-230
North and South America: +1 978 225 2030
California: +1 408 721 1965
Asia-Pacific: +65 6634-9027
China: +86 (21) 64861133

Headquarters

Allied Vision Technologies GmbH
Taschenweg 2a
07646 Stadtroda
Germany

Tel: +49 (0)36428 677-0
Fax: +49 (0)36428 677-28
Managing Directors (Geschäftsführer): Andreas Gerk, Peter Tix

2 Document history and conventions

This chapter includes:

2.1 Document history 10
2.2 Conventions used in this manual 10
2.2.1 Styles 11
2.2.2 Symbols 11

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	2015-11-09	Added USB compatibility, renamed several Vimba components and documents ("AVT" no longer in use), links to new Allied Vision website
1.5	2016-02-27	Added Goldeye CL compatibility, new document layout
1.6	2017-05-01	Added chapter Triggering cameras (including Action Commands), changed the position of `VmbCaptureQueueFlush`, several minor changes, updated document layout
1.7	September 2017	Added some structs and enums, added information to chapter Trigger over Ethernet – Action Commands, updated Troubleshooting, section Goldeye CL cameras, some minor changes
1.7.1	May 2018	Bug fixes
1.8.0	June 2019	Bug fixes, correction of Listing 5
1.8.1	October 2019	Updated for use with GenTL 1.5
1.8.2	May 2020	Bug fixes
1.8.3	October 2020	Added standard-compliant ForceIP features
1.8.4	December 2020	Prepared for use with 5 GigE Vision cameras
1.8.5	May 2021	Several bug fixes, updated some links
1.9.0	October 2021	Added optional "alloc and announce" functionality

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
Emphasis	Programs, or highlighting important things	**Emphasis**
Publication title	Publication titles	Title
Web reference	Links to web pages	Link
Document reference	Links to other documents	Document
Output	Outputs from software GUI	**Output**
Input	Input commands, modes	Input
Feature	Feature names	`Feature`

2.2.2 Symbols

Practical Tip

Safety-related instructions to avoid malfunctions
Instructions to avoid malfunctions

Further information available online

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 (Gigabit Ethernet, 1394, USB, Camera Link). To achieve this generic behavior, Vimba C API utilizes GenICam transport layer modules to connect to the various camera interfaces.

For accessing functionality of either Vimba or the connected cameras, you have two ways of control: You can use the fixed set of API functions and you can use 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.

The Vimba Manual contains a description of the API concepts. To fully understand the API, we recommend reading the Vimba Manual first.

Features are listed in the following documents:

- Allied Vision camera features are described in the Features Reference for your camera.
- Vimba Manual (Vimba System features)

4 API usage

This chapter includes:

4.1 API Version 14
4.2 API Startup and Shutdown 14
4.3 Listing available cameras 15
4.4 Opening and closing a camera 18
4.5 Accessing Features 20
4.6 Image Capture (API) and Acquisition (Camera) 27
4.6.1 Image Capture and Image Acquisition 27
4.6.2 Image Capture 28
4.6.3 Image Acquisition 29
4.7 Using Events 33
4.8 Saving and loading settings 35
4.9 Triggering cameras 36
4.9.1 External trigger 36
4.9.2 Trigger over Ethernet – Action Commands . . . 37
4.10 Additional configuration: Listing Interfaces 39
4.11 Troubleshooting 40
4.11.1 GigE cameras 40
4.11.2 USB cameras 41
4.11.3 Goldeye CL cameras 41
4.12 Error Codes 42

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. Shutting down Vimba API closes all opened cameras.

4.3 Listing available cameras

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)

The order in which the detected cameras are listed is determined by the order of camera discovery and therefore not deterministic. Normally, Vimba recognizes cameras in the following order: USB - 1394 - GigE - Camera Link. However, this order may change depending on your system configuration and the accessories (for example, hubs or long cables).

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.

Camera Link cameras:
The specifications of Camera Link and GenCP do not support plug & play or discovery events. To detect changes to the camera list, shutdown and startup the API by calling `VmbShutdown` and `VmbStartup` consecutively.

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 = (VmbCameraInfo_t*)malloc( nCount * sizeof *pCameras );
        // 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`	Unique identifier for each camera
const char* `cameraName`	Name of the camera
const char* `modelName`	The model name
const char* `serialString`	The serial number
`VmbAccessMode_t` permittedAccess	Access mode, see VmbAccessModeType
const char* `interfaceIdString`	Unique value for each interface or bus

Table 1: Struct `VmbCameraInfo_t`

Similiar to listing available cameras, the function `VmbInterfacesList` can be used to **list available interfaces**, see chapter Additional configuration: Listing Interfaces.

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*`.

`VmbShutdown` blocks until all callbacks have finished execution.

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

- `VmbStartup`
- `VmbShutdown`
- `VmbCameraOpen`
- `VmbCameraClose`
- `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 and 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 (Goldeye CL cameras: configuration only)
- `VmbAccessModeConfig`: enables configuring the IP address of your GigE camera
- `VmbAccessModeRead`: read-only access. Setting features is not possible. However, for GigE cameras that are already in use by another application, the acquired images can be transferred to Vimba API (Multicast).

The enumerations are defined in `VmbAccessModeType` (or its `VmbUint32_t` representation `VmbAccessMode_t`) as shown in Table 2.

Enumeration	Integer Value	Purpose
`VmbAccessModeNone`	0	No access
`VmbAccessModeFull`	1	Read and write access
`VmbAccessModeRead`	2	Read-only access
`VmbAccessModeConfig`	4	Configuration access (GigE)

Table 2: Enum `VmbAccessModeType` is represented as `VmbUint32_t` through `VmbAccessMode_t`

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

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

Vimba API provides several feature types, which all have their specific properties and functionalities, as shown in Table 3.

As shown in Table 3, 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 4. Its referenced data types can be found in Tables 5, 6, and 7. 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.

Information about enumeration features, such as string and integer representation, is held in struct `VmbFeatureEnumEntry_t` as shown in Table 8.

Feature Type	Operation	Function
Enumeration	Set	`VmbFeatureEnumSet`
	Get	`VmbFeatureEnumGet`
	Range	`VmbFeatureEnumRangeQuery`

	Other	`VmbFeatureEnumIsAvailable`
		`VmbFeatureEnumAsInt`
		`VmbFeatureEnumAsString`
		`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 3: Feature types and functions for reading and writing them

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 4: Struct `VmbFeatureInfo_t`

Enumeration	Integer Value	Purpose
`VmbFeatureDataInt`	1	64-bit integer feature
`VmbFeatureDataFloat`	2	64-bit floating point feature
`VmbFeatureDataEnum`	3	Enumeration feature
`VmbFeatureDataString`	4	String feature
`VmbFeatureDataBool`	5	Boolean feature
`VmbFeatureDataCommand`	6	Command feature
`VmbFeatureDataRaw`	7	Raw (direct register access) feature
`VmbFeatureDataNone`	8	Feature with no data

Table 5: Enum `VmbFeatureDataType` is represented as `VmbUint32_t` through `VmbFeatureData_t`

Enumeration	Integer Value	Purpose
VmbFeatureFlagsNone	0	No additional information is provided
VmbFeatureFlagsRead	1	Static info about read access. Current status depends on access mode, check with VmbFeatureAccessQuery()
VmbFeatureFlagsWrite	2	Static info about write access. Current status depends on access mode, check with VmbFeatureAccessQuery()
VmbFeatureFlagsVolatile	8	Value may change at any time
VmbFeatureFlagsModifyWrite	16	Value may change after a write

Table 6: Enum VmbFeatureFlagsType is represented as VmbUint32_t through VmbFeatureFlags_t

Enumeration	Integer Value	Purpose
VmbFeatureVisibilityUnknown	0	Feature visibility is not known
VmbFeatureVisibilityBeginner	1	Feature is visible in feature list (beginner level)
VmbFeatureVisibilityExpert	2	Feature is visible in feature list (expert level)
VmbFeatureVisibilityGuru	3	Feature is visible in feature list (guru level)
VmbFeatureVisibilityInvisible	4	Feature is not visible in feature list

Table 7: Enum VmbFeatureVisibilityType is represented as VmbUint32_t through VmbFeatureVisibility_t

Struct Entry	Purpose
const char* name	Name used in the API
const char* displayName	Enumeration entry name to be used in GUIs
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
VmbInt64_t intValue	Integer value of this enumeration entry

Table 8: Struct VmbFeatureEnumEntry_t

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 = (VmbFeatureInfo_t*)malloc( nCount * sizeof *pFeatures );

    // 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 );
    }
}
```

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", nWidth);
}
```

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;

// 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 9 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 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 9: Basic features found on all cameras

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*`.

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

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)

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

For a quick start, see SynchronousGrab example of the Vimba SDK. For advanced image acquisition including "alloc and announce" (optional, for more efficient buffer allocation), see the AsynchronousGrab example.

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:

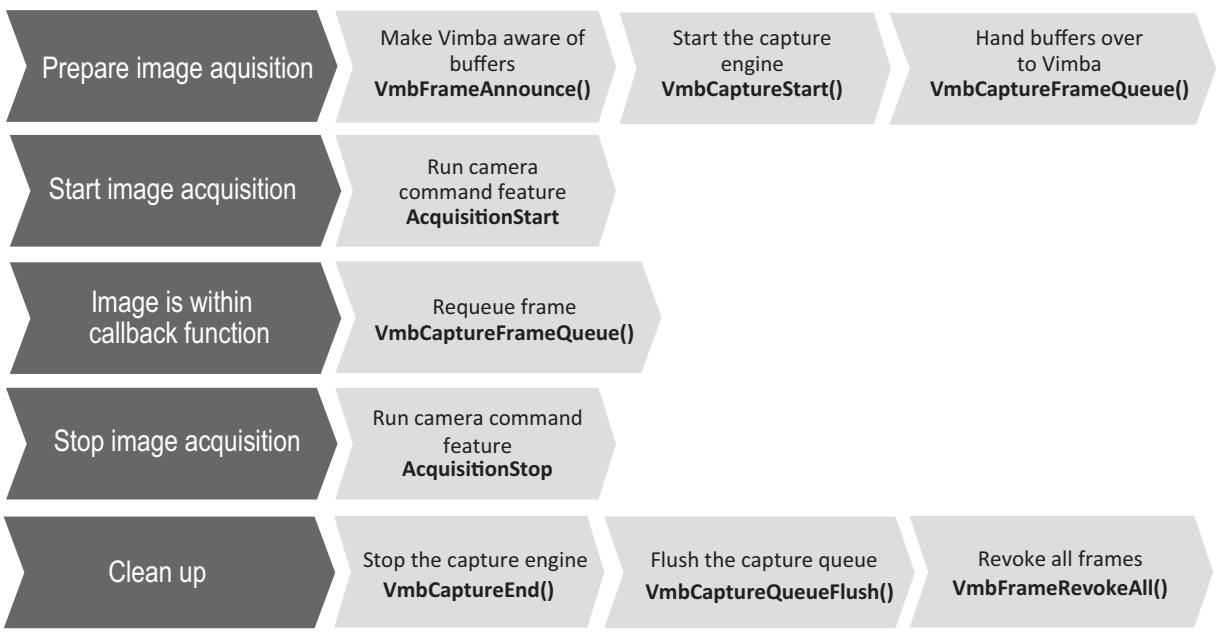


Figure 1: Typical asynchronous application using Vimba C

4.6.2 Image Capture

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). To activate alloc and announce, set the pointer to the buffer to NULL.
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. Stop the capture engine with `VmbCaptureEnd`.
7. Call `VmbCaptureQueueFlush` to cancel all frames on the queue.
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.

Always check that `VmbFrame_t.receiveStatus` equals `VmbFrameStatusComplete` when a frame is returned to ensure the data is valid.

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

```
#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 = malloc( nPLS );                                      (B)
    frames[i].bufferSize = nPLS;                                            (B)
    // Anounce the frame
    // Set frame buffer to NULL to activate alloc and announce
    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)

// Program runtime ...

// When finished, tear down the acquisition chain, close the camera and Vimba
err = VmbFeatureCommandRun( hCamera, "AcquisitionStop" );
err = VmbCaptureEnd( hCamera );
err = VmbCaptureQueueFlush( hCamera );
err = VmbFrameRevokeAll( hCamera );
err = VmbCameraClose( hCamera );
err = VmbShutdown();
```

The struct `VmbFrame_t` represents not only the actual image data, but also additional information as listed in Table 10.

To activate "alloc and announce", set the pointer to the buffer to NULL.

You can find the referenced data types in Tables 11 and 12.

Struct Entry	Type
void* buffer	Pointer to the actual image data (including ancillary data). Can be NULL.
`VmbUint32_t` bufferSize	Size of the data buffer
void* context[4]	4 void pointers that can be employed by the user (e.g. for storing handles)
`VmbFrameStatus_t` receiveStatus	Resulting status of the receive operation
`VmbFrameFlags_t` receiveFlags	Flags indicating which additional frame information is available
`VmbUint32_t` imageSize	Size of the image data inside the data buffer
`VmbUint32_t` ancillarySize	Size of the ancillary data inside the data buffer
`VmbPixelFormat_t` pixelFormat	Pixel format of the image
`VmbUint32_t width`	Width of an image
`VmbUint32_t height`	Height of an image
`VmbUint32_t offsetX`	Horizontal offset of an image
`VmbUint32_t offsetY`	Vertical offset of an image
`VmbUint64_t` frameID	Unique ID of this frame in this stream
`VmbUint64_t` timestamp	Timestamp set by the camera

Table 10: Struct `VmbFrame_t`

Enumeration	Integer Value	Purpose
VmbFrameStatusComplete	0	Frame has been completed without errors
VmbFrameStatusIncomplete	-1	Frame could not be filled to the end
VmbFrameStatusTooSmall	-2	Frame buffer was too small
VmbFrameStatusInvalid	-3	Frame buffer was invalid

Table 11: Enum VmbFrameStatusType is represented as VmbInt32_t through VmbFrameStatus_t

Enumeration	Integer Value	Purpose
VmbFrameFlagsNone	0	No additional information is provided
VmbFrameFlagsDimension	1	Frame's dimension is provided
VmbFrameFlagsOffset	2	Frame's offset is provided (ROI)
VmbFrameFlagsFrameID	4	Frame's ID is provided
VmbFrameFlagsTimestamp	8	Frame's timestamp is provided

Table 12: Enum VmbFrameFlagsType is represented as VmbUint32_t through VmbFrameFlags_t

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 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];
    char callbackReason[255];

    // Get the name of the camera due to which the callback was triggered
    VmbFeatureStringGet( handle, "DiscoveryCameraIdent", cameraName );

    // Get the reason why the callback was triggered. Possible values:
    // Missing (0), a known camera disappeared from the bus
    // Detected (1), a new camera was discovered
    // Reachable (2), a known camera can be accessed
    // Unreachable (3), a known camera cannot be accessed anymore

    VmbFeatureEnumGet( handle, "DiscoveryCameraEvent", callbackReason );
    printf( "Event was fired by camera %s because %s\n", cameraName,
            callbackReason );
}

// 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 );
```

GigE camera events are also handled with the same mechanism of feature invalidation. See Listing 10 for an example.

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
VmbFeatureEnumSet( 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 Saving and loading settings

Additionally to the user sets stored inside the cameras, you can save the feature values as an XML file to your host PC. For example, you can configure your camera with Vimba Viewer, save the settings as a file, and load them with Vimba API. To do this, use the functions `VmbCameraSettingsLoad` and `VmbCameraSettingsSave`.

For a quick start, see example LoadSaveSettings.

To control which features are saved, use the struct listed in Table 13. Note that saving and loading all features including look-up tables may take several minutes. You can manually edit the XML file if you want only certain features to be restored.

Struct Entry	Purpose
`VmbFeaturePersist_t persistType`	Controls which features are to be saved. Valid values are: • VmbFeaturePersistAll: Save all features to XML, including look-up tables • VmbFeaturePersistStreamable: Save only features marked as streamable, excluding look-up tables • VmbFeaturePersistNoLUT: Default, save all features except look-up tables
`Vmbuint32_t maxIterations`	Number of iterations. `LoadCameraSettings` iterates through all given features of the XML file and tries to set each value to the camera. Because of complex feature dependencies, writing a feature value may impact another feature that has already been set by `LoadCameraSettings`. To ensure all values are written as desired, the feature list can be looped several times, given by this parameter. Default value: 5, valid values: 1...10

Table 13: Struct `VmbFeaturePersistSettings_t`

4.9 Triggering cameras

Before triggering, startup Vimba and open the camera(s).

To easily configure the camera's trigger settings, use Vimba Viewer and save/load the settings.

4.9.1 External trigger

The following code snippet shows how to trigger your camera with an external device.

Listing 11: External trigger

```
// Startup Vimba, get cameras and open cameras as usual

// Trigger cameras according to their interface
// Configure trigger input line and selector, switch trigger on
switch( pInterfacetype )
{
case VmbInterfaceEthernet:
    VmbFeatureEnumSet( pCameraHandle, "TriggerSelector", "FrameStart" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerSource", "Line1" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerMode", "On" );
    break;

    // USB: VmbInterfaceUsb

case VmbInterfaceUsb:
    VmbFeatureEnumSet( pCameraHandle, "LineSelector", "Line0" );
    VmbFeatureEnumSet( pCameraHandle, "LineMode", "Input" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerSelector", "FrameStart" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerSource", "Line0" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerMode", "On" );
    break;

case VmbInterfaceFirewire:
    VmbFeatureEnumSet( pCameraHandle, "LineSelector", "Line0" );
    VmbFeatureEnumSet( pCameraHandle, "LineMode", "Input" );
    VmbFeatureEnumSet( pCameraHandle, "LineRouting", "Trigger" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerSelector", "ExposureStart" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerSource", "InputLines" );
    VmbFeatureEnumSet( pCameraHandle, "TriggerMode", "On" );
    break;
}
```

4.9.2 Trigger over Ethernet – Action Commands

Triggering via the AcquisitionStart command (see chapter Image Acquisition) is supported by all cameras. However, it is less precise than triggering with an external device connected to the camera's I/O port.

Selected GigE cameras with the latest firmware additionally support Action Commands. With Action Commands, you can broadcast a trigger signal simultaneously to multiple GigE cameras via GigE cable. Action Commands must be set first to the camera(s) and then to the Vimba API, which sends the Action Commands to the camera(s). As trigger source, select Action0 or Action1.

ActionControl parameters

The following ActionControl parameters must be configured on the camera(s) and then on the host PC.

- `ActionDeviceKey` must be equal on the camera and on the host PC. Before a camera accepts an Action Command, it verifies if the received key is identical with its configured key. Note that `ActionDeviceKey` must be set each time the camera is opened.
  Range (camera and host PC): 0 to 4294967295
- `ActionGroupKey` means that each camera can be assigned to exactly one group for Action0 and a different group for Action1. All grouped cameras perform an action at the same time. If this key is identical on the sender and the receiving camera, the camera performs the assigned action.
  Range (camera and host PC): 0 to 4294967295
- `ActionGroupMask` serves as filter that specifies which cameras within a group react on an Action Command. It can be used to create sub-groups.
  Range (camera): 0 to 4294967295
  Range (host PC): 1 to 4294967295

Executing the API feature `ActionCommand` sends the ActionControl parameters to the cameras and triggers the assigned action, for example, image acquisition. Before an Action Command is executed, each camera validates the received ActionControl parameter values against its configured values. If they are not equal, the camera ignores the command.

More information

For more information about Action Commands, see:

- The ActionCommands programming example of the Vimba SDK
- The application note Trigger over Ethernet - Action Commands
- Listing 12 shows how to send out an Action Command to all connected cameras via all known Gigabit Ethernet interfaces.

Listing 12: Action Commands

```
  // Additionally to this code snippet:
  // Configure the trigger settings and add image streaming

  VmbUint32_t count;
  VmbCameraInfo_t* cameras;
  VmbHandle_t* handles;

  int deviceKey = 11, groupKey = 22, groupMask = 33;

  // Start Vimba and discover GigE cameras
  VmbStartup();

  VmbFeatureBoolGet( gVimbaHandle, "GeVTLIsPresent", &isGigE );
  if( VmbBoolTrue == isGigE )
  {
      VmbFeatureIntSet( gVimbaHandle, "GeVDiscoveryAllDuration", 250 );
      VmbFeatureCommandRun( gVimbaHandle, "GeVDiscoveryAllOnce" );
  }

  // Get cameras
  VmbCamerasList( NULL, 0, &count, sizeof(*cameras) );
  cameras = (VmbCameraInfo_t *) malloc( count * sizeof(*cameras) );
  VmbCamerasList( cameras, count, &count, sizeof(*cameras) );

  // Allocate space for handles
  handles = (VmbHandle_t*) malloc( count * sizeof(VmbHandle_t) );

  for( int i=0; i<count; ++i )
  {
      const char* cameraId = cameras[i].cameraIdString;

      // Open camera
      VmbCameraOpen( cameraId, VmbAccessModeFull, &handles[i] );

      // Set device key, group key and group mask
      // Configure trigger settings (see programming example)
      VmbFeatureIntSet( handles[i], "ActionDeviceKey", deviceKey );
      VmbFeatureIntSet( handles[i], "ActionGroupKey", groupKey );
      VmbFeatureIntSet( handles[i], "ActionGroupMask", groupMask );
  }

  // Set Action Command to API
  // Allocate buffers and enable streaming (see programming example)
  VmbFeatureIntSet( gVimbaHandle, "ActionDeviceKey", deviceKey );
  VmbFeatureIntSet( gVimbaHandle, "ActionGroupKey", groupKey );
  VmbFeatureIntSet( gVimbaHandle, "ActionGroupMask", groupMask );

  // Send Action Command
  VmbFeatureCommandRun( gVimbaHandle, "ActionCommand" );

// If no further Actions will be applied: close cameras, shutdown API, and
// free allocated space as usual
```

4.10 Additional configuration: Listing Interfaces

`VmbInterfacesList` enumerates all Interfaces (GigE, USB, or 1394 adapters, or Camera Link frame grabbers) recognized by the underlying transport layers.
See Listing 13 for an example.

Listing 13: Get Interfaces

```
VmbUint32_t nCount;
VmbInterfaceInfo_t *pInterfaces;

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

// Allocate accordingly
pInterfaces = (VmbInterfaceInfo_t*)malloc( nCount * sizeof *pInterfaces );

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

Struct `VmbInterfaceInfo_t` provides the information about an interface as listed in Table 14.

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
`VmbAccessMode_t` permittedAccess	The mode to open the interface

Table 14: 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*`.

`VmbShutdown` blocks until all callbacks have finished execution.

Enumeration	Integer Value	Purpose
`VmbInterfaceUnknown`	0	Interface is not known to this version of the API
`VmbInterfaceFirewire`	1	IEEE 1394
`VmbInterfaceEthernet`	2	GigE
`VmbInterfaceUsb`	3	USB
`VmbInterfaceCL`	4	Camera Link

Table 15: Enum `VmbInterfaceType` is represented as `VmbUint32_t` through `VmbInterfaceInfo_t`

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

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

4.11 Troubleshooting

4.11.1 GigE cameras

To get your 5 GigE Vision camera up and running, see the User Guide for your camera.

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.
- DeviceThroughputLimit (legacy name: 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:

Please find detailed installation instructions in the User Guide for your camera.

4.11.2 USB cameras

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

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

4.11.3 Goldeye CL cameras

- The pixel format, all features affecting the image size, and DeviceTapGeometry must be identical in Vimba and the frame grabber software.
- Make sure to select an image size supported by the frame grabber.
- The baud rate of the camera and the frame grabber must be identical.

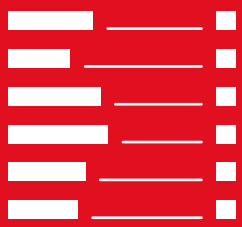
4.12 Error Codes

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 16 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	The current operation was not completed (e.g. a multiple registers read or write)
VmbErrorIO	-20	There was an error during read or write with devices (camera or disk)

Table 16: Error codes returned by Vimba

5 Function reference



This chapter includes:

5.1 Callbacks 45
5.2 API Version 46
5.3 API Initialization 47
5.4 Camera Enumeration & Information 48
5.5 Features 51
5.6 Integer 55
5.7 Float 58
5.8 Enum 61
5.9 String 66
5.10 Boolean 68
5.11 Command 70
5.12 Raw 71
5.13 Feature invalidation 74
5.14 Image preparation and acquisition 76
5.15 Interface Enumeration & Information 80
5.16 Ancillary data 82
5.17 Memory/Register access 83

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

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

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:** If "pVersionInfo" is NULL.

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

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

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

5.3.2 VmbShutdown()

Perform a shutdown on the API.

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:** If "pNumFound" was NULL

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:** If "idString" was NULL

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 "DEV_1234567890" for an ID as reported by Vimba

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

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 "DEV_1234567890" for an ID as reported by Vimba

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

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

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

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

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

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:** If "pIsReadable" and "pIsWriteable" were both NULL
- **VmbErrorNotFound:** The feature was not found

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:** If "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:** If "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

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	`VmbBool_t *`	pHasIncrement	"true" if this float feature has an increment.
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:** If "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**`	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; may 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" or "pIntVal" is NULL

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

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. May 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

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:** If 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:** If "value" is not within valid bounds
- **VmbErrorNotFound:** If the feature is not found

- **VmbErrorBadParameter:** If "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:** If "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:** If "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

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

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

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
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

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

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

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. The method can be used to annouce a previously allocated frame buffer to the transport layer. Alternatively, in case "pFrame->buffer" points to NULL, the method will allocate and announce a new buffer. In this case "pFrame->buffer" contains the allocated buffer address on return.

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

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

Consequences of VmbCaptureEnd(): - The frame callback will not be called anymore

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

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

Control of all the currently queued frames will be returned to the user, leaving no frames in the capture 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:** If "pNumFound" was NULL

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:** If "pInterfaceHandle" was NULL

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

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:** No error
- **VmbErrorBadHandle:** Chunk mode of the camera was not activated. See feature ChunkModeActive
- **VmbErrorApiNotStarted:** VmbStartup() was not called before the current command

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

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

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

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.

5.17.5 VmbCameraSettingsSave()

Saves all feature values to XML file.

	Type	Name	Description
in	`const VmbHandle_t`	handle	Handle for an entity that allows register access
in	`const char*`	fileName	Name of XML file to save settings
in	`VmbFeaturePersistSettings_t*`	pSettings	Settings struct
in	`VmbUint32_t`	sizeofSettings	Size of settings struct

- **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 "fileName" is NULL

Camera must be opened beforehand and function needs corresponding handle. With given filename parameter path and name of XML file can be determined. Additionally behaviour of function can be set with providing 'persistent struct'.

5.17.6 VmbCameraSettingsLoad()

Load all feature values from XML file to device.

	Type	Name	Description
in	`const VmbHandle_t`	handle	Handle for an entity that allows register access
in	`const char*`	fileName	Name of XML file to save settings
in	`VmbFeaturePersistSettings_t*`	pSettings	Settings struct
in	`VmbUint32_t`	sizeofSettings	Size of settings struct

- **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 "fileName" is NULL

Camera must be opened beforehand and function needs corresponding handle. With given filename parameter path and name of XML file can be determined. Additionally behaviour of function can be set with providing 'settings struct'.