Vimba C API

Programmer's Manual

V1.2
2013-Jun-25

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

ALLIED Vision Technologies

Legal Notice

Trademarks

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

Warranty

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

Copyright

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

Allied Vision Technologies GmbH 06/2013

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

Headquarters:

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

Contents

Listings

1 Contacting Allied Vision Technologies

Note

- **Technical Information**
  http://www.alliedvisiontec.com
- **Support**
  support@alliedvisiontec.com

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

Allied Vision Technologies Canada Inc.
101-3750 North Fraser Way
Burnaby, BC, V5J 5E9, Canada
Tel: +1 604-875-8855
Fax: +1 604-875-8856
Email: info@alliedvisiontec.com

Allied Vision Technologies Inc.
38 Washington Street
Newburyport, MA 01950, USA
Toll Free number +1 877-USA-1394
Tel.: +1 978-225-2030
Fax: +1 978-225-2029
Email: info@alliedvisiontec.com

Allied Vision Technologies Asia Pte. Ltd.
82 Playfair Road
#07-02 D'Lithium
Singapore 368001
Tel. +65 6634-9027
Fax:+65 6634-9029
Email: info@alliedvisiontec.com

Allied Vision Technologies (Shanghai) Co., Ltd.
2-2109 Hongwell International Plaza
1602# ZhongShanXi Road
Shanghai 200235, China
Tel: +86 (21) 64861133
Fax: +86 (21) 54233670
Email: info@alliedvisiontec.com

2 Introduction

2.1 Document history

Version	Date	Changes
1.0	2012-Nov-15	Initial version
1.1	2013-Feb-22	Different links, small changes
1.2	2013-Jun-18	Small corrections, layout changes

2.2 Conventions used in this manual

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

2.2.1 Styles

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

2.2.2 Symbols

Note

This symbol highlights important information.

Caution

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

www

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

3 General aspects of the API

The purpose of AVT Vimba APIs is to enable programmers to interact with AVT cameras independent of the interface technology (1394, Gigabit Ethernet). To achieve this, Vimba API utilizes different transport layers to connect to the various camera interfaces and is therefore considered generic in terms of camera interfaces. For accessing functionality of either Vimba or the connected cameras, you have two ways of control: the generic functions on the one hand and feature access to Vimba, the transport layers, and the cameras on the other. This manual deals only with the functional part.

4 Module Version

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

5 Module Initialization

Using Vimba API always begins with a call to `VmbStartup`. Before using any Vimba API functions (other than `VmbVersionQuery`), you must initialize the API with this call. When you have finished using Vimba API, call `VmbShutdown` to free resources. These two API functions must always be paired. It is possible, although not recommended, to call the pair several times within the same program.

6 List available cameras

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

`VmbCamerasList` will enumerate all cameras recognized by the underlying transport layers. With this command, the programmer can fetch all static details of a camera such as its ID, its model and vendor name or the ID of the interface (e.g. the network or 1394 adapter) it is connected to. For 1394 cameras this attempt is straightforward, as opposed to GigE; due to its asynchronous nature, listing cameras over the network is a two-step process. First a device discovery request has to be sent out before Vimba API can be aware of all GigE devices that answered that request. Vimba API puts the developer in charge of deciding how to send out discovery packets. However, this can be achieved through the command features *GeVDiscoveryAllOnce* and *GeVDiscoveryAllAuto*, whereby the latter constantly emits discovery commands. To stop discovery, use the command feature *GeVDiscoveryAllNone*. Note that these features can be applied to all network interfaces as well as to one particular interface only. See Listing 1 for an example.

Listing 1: Get Cameras

```
1   bool bGigE;
2   VmbUint32_t nCount;
3   VmbCameraInfo_t *pCameras;
4
5   // We ask Vimba for the presence of a GigE transport layer
6   VmbError_t err = VmbFeatureBoolGet( gVimbaHandle, "GeVTLIsPresent", &bGigE );
7   if ( VmbErrorSuccess == err )
8   {
9       if ( true == bGigE )
10      {
11          // We query all network interfaces using the global Vimba handle
12          err = VmbFeatureCommandRun( gVimbaHandle, "GeVDiscoveryAllOnce" );
13          // Wait for the discovery packets to return
14          Sleep( 200 );
15      }
16  }
17  if ( VmbErrorSuccess == err )
18  {   // Get the amount of connected cameras
19      err = VmbCamerasList( NULL, 0, &nCount, sizeof *pCameras );
20
21      if ( VmbErrorSuccess == err )
22      {
23          // Allocate accordingly
24          pCameras = new VmbCameraInfo_t[ nCount ];
25          // Get the cameras
26          err = VmbCamerasList( pCameras, nCount, &nCount, sizeof *pCameras );
27          // Print out each camera's name
28          for ( VmbUint32_t i=0; i<nCount; ++i )
29          {
30              std::cout << pCameras[i].cameraName << std::endl;
31          }
32      }
33  }
```

The `VmbCameraInfo_t` struct provides the entries listed in Table 1 for obtaining information about a camera.

To get notified whenever a camera is detected, disconnected, or changes its open state, use `VmbFeatureInvalidationRegister` to register a callback that gets executed on the according event.

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

Table 1: VmbCameraInfo_t struct

Use the global Vimba handle for registration. The function pointer to the callback function has to be of type `VmbInvalidationCallback*`. Note that the continuous sending of discovery packages has to be turned on to enable Vimba to recognize GigE camera events. Please note that `VmbShutdown` blocks until all callbacks have finished execution. Below you find a list of functions that cannot be called within the callback routine.

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

7 Opening a camera

A camera must be opened to control it and to capture images. To open a camera, call `VmbCameraOpen` and provide the ID of the camera as well as the desired access mode. GigE cameras may also be identified by their IP or MAC address. 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

```
1   VmbCameraInfo_t *pCameras;
2   VmbHandle_t hCamera;
3
4   // Get all known cameras as described in chapter "List available cameras"
5
6   // Open the first camera
7   if ( VmbErrorSuccess == VmbCameraOpen(  pCameras[0].cameraIdString,
8                                           VmbAccessModeFull, hCamera ) )
9   {
10      std::cout << "Camera opened, handle [" << hCamera << "] retrieved.";
11  }
```

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

Listing 3: Close Camera

```
1   if ( VmbErrorSuccess == VmbCameraClose( hCamera ) )
2   {
3       std::cout << "Camera closed." << std::endl;
4   }
```

8 Feature Access

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

GenICam-compliant features control and monitor various aspects of the drivers and cameras. For more details on features see the Vimba SDK Features, the 1394 Transport Layer Feature Description or the GigE Vision Transport Layer Feature Description.

There are several feature types which have type-specific properties and allow type-specific functionality: Integer, Float, Enum, String, Boolean, Raw data. Additionally, since not all the features are available all the time, there is a general necessity for querying the accessibility of features. Vimba API provides its own set of access functions for every feature data type. The data type of a feature as well as additional static properties of a feature are held in the `VmbFeatureInfo_t` struct.

To start continuous acquisition, set the feature *AcquisitionMode* to *Continuous* and run the command feature *AcquisitionStart* as shown in Listing 4.

Listing 4: Acquisition Start

```
VmbHandle_t hCamera;

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

if ( VmbErrorSuccess == VmbFeatureEnumSet(  hCamera, "AcquisitionMode",
                                            "Continuous" ))
{
    if ( VmbErrorSuccess = VmbFeatureCommandRun( hCamera, "AcquisitionStart" ))
    {
        std::cout << "Acquisition successfully started" << std::endl;
    }
}
```

To read the image size in bytes, see Listing 5.

Listing 5: Payload Size

```
VmbHandle_t hCamera;

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

VmbInt64_t nPayloadSize;

if ( VmbErrorSuccess == VmbFeatureIntGet(   hCamera, "PayloadSize",
                                            &nPayloadSize ))
{
    std::out << nPayloadSize << std::endl;
}
```

To simply query all available features of a camera, use `VmbFeaturesList`. This list does not change while the camera is opened as shown in Listing 6.

Listing 6: Get Features

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

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

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

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

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

    // Print out their name and data type
    for ( int i=0; i<nCount; ++i )
    {
        std::cout << "Feature " << pFeatures[i].name;
        std::cout << " of type: " << pFeatures[i].featureDataType << std::endl;
    }
}
```

Table 2 introduces 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 2: Basic features found on all cameras

Make sure to set the *PacketSize* feature of GigE cameras to a value supported by your network card. The command feature *GVSPAdjustPacketSize* configures GigE cameras to use the largest possible packets. Please note that the automatic adjustment might not lead to the expected results in a multiple camera scenario (many cameras connected to one GigE interface). Here the available bandwidth has to be shared between all cameras. See the feature *StreamBytesPerSecond* for this. Furthermore, the maximum packet

size might not be available by all connected cameras. If you experience problems streaming even when the available bandwidth is equally shared between all cameras, try to reduce the packet size.

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*`. Please note that `VmbShutdown` only returns after all callbacks have finished execution. Below you find a list of functions that cannot be called within the callback routine.

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

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: Functions for reading and writing a Feature

9 Image Acquisition and Capture

For a quick start see SynchronousGrab example of the Vimba SDK.
To obtain an image from your camera, first setup Vimba API to capture images, then start the acquisition on the camera. These two concepts – capture and acquisition – while related, are independent operations as it is shown below (the bracketed tokens refer to the example at the end of this chapter).
To capture images sent by the camera, follow these steps:

1. `VmbFrameAnnounce` – Make a frame known to the API so that it can allocate internal resources (1).
2. `VmbCaptureStart` – Start the capture engine of the API. Prepare the capture stream (2).
3. `VmbCaptureFrameQueue` – Queue (an already announced) frame. As images arrive from the camera, they are placed in the next frame's buffer in the queue, and returned to the user (3).
4. When done, `VmbCaptureEnd` – Stop the capture engine and close the image capture stream.
5. If frames have been announced before, call `VmbFrameRevokeAll` eventually.

None of the steps above have a direct effect on the camera. To start image acquisition, follow these steps:

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

To stop image acquisition, run command feature *AcquisitionStop*.
Normally, image capture is initialized and frame buffers are queued before the command *AcquisitionStart* is run, but the order can vary depending on the application. To guarantee that a particular image is captured, ensure that the frames are queued before *AcquisitionStart*.

9.1 Image Capture

Images are captured using frame buffers that are given to Vimba in calls to the asynchronous function `VmbCaptureFrameQueue` (3). As long as the frame queue holds a frame whose buffer is large enough to contain the image data, it is filled with the incoming image. Allocating a frame's buffer is left to the API, although it is possible to allocate a piece of memory yourself that you then pass to the API. In both cases first query the needed amount of memory through the feature *PayloadSize* (A) or calculate it yourself. Allocate memory according to the payload size, declare a `VmbFrame_t` and let its buffer point to this block of memory (B). After that, announce the frame (1), start the capture engine (2), and queue the frame you have just created with `VmbCaptureFrameQueue` (3), so it can be filled when acquisition has started.
Before a queued frame can be used or modified, the application needs to know when the image capture is complete. Two mechanisms are available: either block your thread until capture is complete using `VmbCaptureFrameWait` for just a single image, or 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. Below, you find a list of functions that cannot be called within the callback routine.

- `VmbStartup`

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

NOTE: Always check that `VmbFrame_t.receiveStatus` equals `VmbFrameStatusComplete` when a frame is returned to ensure the data is valid.
Many frames can be placed on the frame queue, and their image buffers will be filled in the same order they were queued. To capture more images, keep submitting new frames (frames that you have processed can be re-queued) as the old frames complete. Most applications need not queue more than two or three frames at a time.
If you want to cancel all the frames on the queue, call `VmbCaptureQueueFlush`.

9.2 Image Acquisition

Image acquisition is set up with the features *AcquisitionMode*, AcquisitionStart (4). For stopping acquisition, feature Acquisition is normally used.
Listing 7 shows a minimal streaming example (without error handling for the sake of simplicity).

Listing 7: Streaming

```
#define FRAME_COUNT 3            // We choose to use 3 frames
VmbError_t err;                  // Every Vimba function returns an error code that the
                                 // programmer should always 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 )
    {
        std::cout << "Frame successfully received" << std::endl;
    }
    else
    {
        std::cout << "Error receiving frame" << std::endl;
    }
    VmbCaptureFrameQueue( hCamera, pFrame, FrameDoneCallback );
}

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

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

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

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

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

10 Additional configuration: List available interfaces

VmbInterfacesList will enumerate all interfaces (GigE or 1394 adapters) recognized by the underlying transport layers.
See Listing 8 for an example.

Listing 8: Get Interfaces

```
1   VmbUint32_t nCount;
2   VmbInterfaceInfo_t *pInterfaces;
3
4   // Get the amount of connected interfaces
5   VmbInterfacesList( NULL, 0, &nCount, sizeof *pInterfaces );
6
7   // Allocate accordingly
8   pInterfaces = new VmbInterfaceInfo_t[ nCount ];
9
10  // Get the interfaces
11  VmbInterfacesList( pCameras, nCount, &nCount, sizeof *pInterfaces );
```

The VmbInterfaceInfo_t struct provides the information about an interface as listed in Table 4.

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 4: VmbInterfaceInfo_t struct

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*. Please note that VmbShutdown blocks until all callbacks have finished execution. Below you find a list of functions that cannot be called within the callback routine.

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

11 Error Codes

All Vimba API functions return an error code of type `VmbErrorType`.
Typical errors are listed with each function in the Vimba C Function Reference Manual. However, any of the error codes listed in Table 5 might be returned.

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

Table 5: Error codes returned by Vimba

12 Function reference

For a complete list of all methods, see the Vimba C Function Reference Manual