AVT SOFTWARE DEVELOPMENT KIT

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

1 Contacting Allied Vision Technologies 5

2 Introduction 6
- 2.1 Document history . . . 6
- 2.2 Conventions used in this manual . . . 6
  - 2.2.1 Styles . . . 6
  - 2.2.2 Symbols . . . 6

3 General aspects of the API 7

4 Module Version 7

5 Module Initialization 7

6 Shared Pointers 8
- 6.1 General aspects . . . 8
- 6.2 Restrictions . . . 8
- 6.3 Customizing shared pointer usage . . . 8

7 List available cameras 10

8 Opening a camera 12

9 Feature Access 13

10 Image Acquisition and Capture 17
- 10.1 Image Capture . . . 17
- 10.2 Image Acquisition . . . 18

11 Additional configuration: List available interfaces 20

12 Error Codes 21

13 Function reference 22

Listings

1 Shared Pointer 8
2 Get Cameras 10
3 Open Camera 12
4 Open Camera by IP 12
5 Acquisition Start 13
6 Payload Size 13
7 Streaming 19
8 Get Interfaces 20

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-16	Initial version
1.1	2013-Mar-05	Minor corrections, added info about what functions can be called in which callback
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

AVT Vimba C++ API is an object orientated C++ API. It utilizes different transport layers to connect to the various camera interfaces (FireWire, Gigabit Ethernet) and is therefore considered generic in terms of camera interfaces. Vimba API makes intense use of shared pointers to ease object lifetime and memory allocation. Vimba API relieves the developer of all memory allocations. Vimba API is equipped with a shared pointer implementation, because some C++ runtime libraries don't provide one. This means it is your choice which shared pointer implementation you prefer to use. Beside `std::shared_ptr` and Vimba's own implementation, the API can be freely configured to use any other shared pointer class such as `boost::shared_ptr` or `QSharedPointer` from the Qt library.

4 Module Version

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

5 Module Initialization

The entry point to Vimba API is the `VimbaSystem` singleton. Use `VimbaSystem::GetInstance` to obtain a reference to it. The `VimbaSystem` object allows both, to control the API's behavior and to query for interfaces and cameras. Before calling any Vimba API functions (other than `VimbaSystem::GetInstance` and `VimbaSystem::QueryVersion`), the API must be initialized by calling `VimbaSystem::Startup` through the singleton. When you are finished with Vimba API, call `VimbaSystem::Shutdown` 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. Successive calls of `VimbaSystem::Startup` or `VimbaSystem::Shutdown` are ignored. Therefore the first `VimbaSystem::Shutdown` after a `VimbaSystem::Startup` will close the API.

6 Shared Pointers

6.1 General aspects

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

Listing 1: Shared Pointer

```
{
    // This declares an empty shared pointer that can wrap a pointer of
    // type Camera
    CameraPtr sp1;

    // The reset member function tells the shared pointer to
    // wrap the provided raw pointer
    // sp1 now has a reference count of 1
    sp1.reset( new Camera() );
    {
        // In this new scope we declare another shared pointer
        CameraPtr sp2;

        // By assigning sp1 to it the reference count of both (!) is set to 2
        sp2 = sp1;
    }
    // When sp2 goes out of scope the reference count drops back to 1
}
// Now that sp1 has gone out of scope its reference count has dropped
// to 0 and it has released the underlying raw pointer on destruction
```

6.2 Restrictions

Unfortunately, shared pointers (or smart pointers in general) were not part of the C++ standard library until C++11. For example, the first version of Microsoft's C++ standard library implementation that supports shared pointers is included in Visual Studio 2010. Due to the aforementioned circumstances, Vimba C++ API makes heavy use of shared pointers while not relying on a specific implementation. You can easily replace the shared pointer type that Vimba uses by default with your own. Furthermore, Vimba C++ API comes with its built-in shared pointer type in case your C++ library does not provide one.

6.3 Customizing shared pointer usage

The steps to follow to use a custom shared pointer type in Vimba:

1. Add the define USER_SHARED_POINTER to your compiler settings
2. Add your shared pointer source files to the Vimba C++ API project

3. Define the macros and typedefs as described in the header UserSharedPointerDefines.h

The define USER_SHARED_POINTER tells Vimba to include a header file named UserSharedPointerDefines.h in which several typedefs and macros are defined. Table 1 lists these macros covering the basic functionality that Vimba expects from any shared pointer. Since a shared pointer is a generic type, it requires a template parameter. That is what the various typedefs are for. For example, the `CameraPtr` is just an alias for `AVT::VmbAPI::shared_ptr<AVT::VmbAPI::Camera>`.

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

Table 1: Basic functions of a shared pointer class

After you have completed these steps and recompiled Vimba C++ API, Vimba is ready to use the provided shared pointer implementation without changing its behavior. Within your own application, you can employ your shared pointers as usual. Please note that your application and Vimba have to refer to the very same shared pointer type. If you want your application to substitute its shared pointer type along with Vimba, feel free to utilize the macros listed in Table 1 in your application as well.

7 List available cameras

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

`VimbaSystem::GetCameras` will enumerate all cameras recognized by the underlying transport layers. See Listing 2 for an example.

Listing 2: Get Cameras

```
std::string name;
CameraPtrVector cameras;
VimbaSystem &system = VimbaSystem::GetInstance();

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

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

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

Table 2: Basic functions of Camera class

Static features that do not change throughout the object's lifetime such as ID and Name can be queried without having to open the camera. To get notified when a camera is detected or disconnected, use `VimbaSystem::RegisterCameraListObserver`. The observer to be registered has to implement the interface `ICameraListObserver`. This interface declares the member function `CameraListChanged`. In your implementation of this function, you can react on cameras being plugged in or out as it will get called by Vimba API on the according event. Please note that `VimbaSystem::Shutdown` blocks until all

callbacks have finished execution. Below, you find a list of functions that cannot be called within the callback routine.

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

8 Opening a camera

A camera must be opened for control and to capture images. To open a camera, simply call `Camera::Open`. If you already know the ID of a camera (GigE cameras can also be identified by their IP or MAC address), call `VimbaSystem::OpenCameraByID`. An example for opening a camera retrieved from the camera list is shown in Listing 3.

Listing 3: Open Camera

```
CameraPtrVector cameras;
VimbaSystem &system = VimbaSystem::GetInstance();

if ( VmbErrorSuccess == system.Startup() )
{
    if ( VmbErrorSuccess == system.GetCameras( cameras ) )
    {
        for (   CameraPtrVector::iterator iter = cameras.begin();
                cameras.end() != iter;
                ++iter )
        {
            if ( VmbErrorSuccess == (*iter)->Open( VmbAccessModeFull ) )
            {
                std::cout << "Camera opened" << std::endl;
            }
        }
    }
}
```

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

Listing 4: Open Camera by IP

```
CameraPtr camera;
VimbaSystem &system = VimbaSystem::GetInstance();

if ( VmbErrorSuccess == system.Startup() )
{
    if ( VmbErrorSuccess == system.OpenCameraByID(  "192.168.0.42",
                                                    VmbAccessModeFull,
                                                    camera ) )
    {
        std::cout << "Camera opened" << std::endl;
    }
}
```

To close a camera use `Camera::Close`.

9 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.

To start continuous acquisition, set the feature *AcquisitionMode* to *Continuous* and run the command feature *AcquisitionStart* as shown in Listing 5.

Listing 5: Acquisition Start

```
FeaturePtr feature;

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

To read the image size in bytes, see Listing 6.

Listing 6: Payload Size

```
FeaturePtr feature;
VmbInt64_t payloadSize;

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

Table 3 introduces the basic features of all cameras. A feature has a name, a type, and access flags such as `read-permitted` and `write-permitted`.

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

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 3: Basic features found on all cameras

size might not be available to 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.
With `Camera::GetFeatures`, you can list all features available for a camera. This list remains static while the camera is opened. The `Feature` class provides the feature's value and further information. Use the following member functions of class `Feature` to access these:

`GetName( std::string& )` Name of the feature

`GetDisplayName( std::string& )` Name to display in GUI

`GetDataType( VmbFeatureDataType& )` Data type of the feature

`GetFlags( VmbFeatureFlagsType& )` Special flags

`GetCategory( std::string& )` Category the feature belongs to

`GetPollingTime( VmbUint32_t& )` The suggested time to poll the feature

`GetUnit( std::string& )` The unit of the feature if available

`GetRepresentation( std::string& )` The scale to represent the feature

`GetVisibility( VmbFeatureVisibilityType& )` The audience the feature is for

`GetToolTip( std::string& )` Short description of the feature

`GetDescription( std::string& )` Description of the feature

`GetSFNCNamespace( std::string& )` The SFNC namespace of the feature

`GetAffectedFeatures( FeaturePtrVector& )` Features that change if the feature is changed

`GetSelectedFeatures( FeaturePtrVector& )` Features that are selected by the feature

`IsReadable( bool& )` Determines if read access will succeed

`IsWritable( bool& )` Determines if write access will succeed

The `GetDataType` function gives information about the available functions for the feature. Table 4 lists the available functions depending on the type returned by `GetDataType`. Every function a particular feature type does not support will return `VmbErrorWrongType`.

`GetDisplayName` gives the feature name to be used in GUI text elements. `GetToolTip` and `GetDescription` provide text for bubble help and extended help functionality. `GetSFNCNamespace`, `GetCategory` and `GetVisibility` can be used to filter and group feature representation, e.g. in a tree view. `GetRepresentation` can be used to change behavior of sliders for a feature, where `GetUnit` can provide a unit to be displayed as additional user information.

`GetFlags` gives information about the actions available for a feature and how changes might affect the feature. `Read` and `Write` flags (also available as member functions `IsReadable` and `IsWritable`) determine whether get and set functions will succeed. `Volatile` features cannot be expected to return the same value in successive reads. `ModifyWrite` features will adjust values by setting them to valid values.

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

With the member function `SetValue`, a feature's value can be set. Table 4 lists the Vimba API functions of the `Feature` class used to access feature values.

Feature Type	Set	Get	Range	Other
Enum	SetValue(string)	GetValue(string)	GetValues(StringVector)	IsValueAvailable(string)
Enum	SetValue(int)	GetValue(int)	GetValues(IntVector)	IsValueAvailable(int)
Enum		GetEntry(EnumEntry)	GetEntries(EntryVector)	
Int64	SetValue(int)	GetValue(int)	GetRange(int, int)	GetIncrement(int)
Float	SetValue(double)	GetValue(double)		
String	SetValue(string)	GetValue(string)		
Bool	SetValue(bool)	GetValue(bool)		
Command	RunCommand	IsCommandDone		
Raw	SetValue(uchar)	GetValue(uchar)		

Table 4: Functions for reading and writing a Feature

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

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

To get notified when a feature's value changes use `Feature::RegisterObserver`. The observer to be registered has to implement the interface `IFeatureObserver`. This interface declares the member function `FeatureChanged`. In the implementation of this function, you can react on updated feature

values as it will get called by Vimba API on the according event. Please note that `VimbaSystem::Shutdown` blocks until all callbacks have finished execution. Below, you find a list of functions that cannot be called within the callback routine.

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

10 Image Acquisition and Capture

For a quick start see SynchronousGrab, AsynchronousGrab or SampleViewer examples of the Vimba SDK.
To obtain an image from your camera, first set up 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. `Camera::AnnounceFrame` – Make a frame known to the API so that it can allocate internal resources (1).
2. `Camera::StartCapture` – Start the capture engine of the API. Prepare the capture stream (2).
3. `Camera::QueueFrame` – 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, `Camera::EndCapture` – Stop the capture engine and close the image capture stream.

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 a particular image is captured, you must ensure that your frames are queued before *AcquisitionStart*.

10.1 Image Capture

Images are captured using frame buffers that are given to Vimba in calls to the asynchronous function `Camera::QueueFrame` (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 into the API. In both cases first query the needed amount of memory through the feature *PayloadSize* (A) or calculate it yourself. Then create a `Frame` object and pass either the size of desired memory or a pointer to already allocated memory to the constructor (B). After that, announce the frame (1), start the capture engine (2), and queue the frame you have just created with `Camera::QueueFrame` (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 `Camera::AcquireSingleImage` for just a single image or `Camera::AcquireMultipleImages` for many images, or register an observer with `Frame::RegisterObserver` (C). The observer to be registered has to implement the interface `IFrameObserver`. In its working routine `IFrameObserver::FrameReceived`, you can implement your frame handling code as well as queue the frame again after you have processed it. This working routine is called when image capture is complete. Below, you find a list of functions that cannot be called within the callback routine.

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

NOTE: Always check that Frame::GetReceiveStatus returns 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 Camera::Flush. In case the API has done memory allocation, this memory is not released until the camera class' FlushQueue, RevokeAllFrames / RevokeFrame, EndCapture or Close function has been called.

10.2 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

```
VmbErrorType err;    // Every Vimba function returns an error code that the
                     // programmer should always check for VmbErrorSuccess
VimbaSystem &sys;    // A reference to the VimbaSystem singleton
CameraPtrVector cameras;     // A list of known cameras
FramePtrVector frames( 3 ); // A list of frames for streaming. We chose
                            // to queue 3 frames.
IFrameObserverPtr pObserver( new MyFrameObserver() );   // Our implementation
                                                        // of a frame observer
FeaturePtr pFeature;         // Any camera feature
VmbUInt64_t nPLS;            // The payload size of one frame

sys = VimbaSystem::GetInstance();

err = sys.GetCameras( cameras );

err = cameras[0]->Open( VmbAccessModeFull );

err = cameras[0]->GetFeatureByName( "PayloadSize", pFeature );              (A)
err = pFeature->GetIntValue( nPLS )                                         (A)

for (   FramePtrVector::iterator iter = frames.begin();
        frames.end() != iter;
        ++iter )
{
    ( *iter )->reset( new Frame( nPLS ) );                                  (B)
    err = ( *iter )->RegisterObserver( pObserver ) );                       (C)
    err = cameras[0]->AnnounceFrame( *iter  );                              (1)
}

err = StartCapture();                                                       (2)

for (   FramePtrVector::iterator iter = frames.begin();
        frames.end() != iter;
        ++iter )
{
    err = cameras[0]->QueueFrame( *iter  );                                 (3)
}

err = GetFeatureByName( "AcquisitionStart", pFeature );                     (4)
err = pFeature->RunCommand();                                               (4)
```

11 Additional configuration: List available interfaces

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

Listing 8: Get Interfaces

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

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

The Interface class provides the member functions to obtain information about an interface listed in Table 5.

Function	Purpose
VmbErrorType GetID(std::string&) const	The unique ID
VmbErrorType GetName(std::string&) const	The name
VmbErrorType GetType(VmbInterfaceType&) const	The camera interface type
VmbErrorType GetSerialNumber(std::string&) const	The serial number
VmbErrorType GetPermittedAccess(VmbAccessModeType&) const	The mode to open the interface

Table 5: Basic functions of Interface class

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

12 Error Codes

All Vimba API functions return an error code of type `VmbErrorType`.
Typical errors are listed with each function in the Function Reference Manual. However, any of the error codes listed in Table 6 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 6: Error codes returned by Vimba

13 Function reference

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