Basler ToF Driver Package 1.3.2

C++ Programmer's Guide

BASLER
the power of sight

Contents

Chapter 1

Introduction

The ToF Programmer's Guide is a quick guide on how to write program code for the Basler ToF Camera using the Basler ToF C++ API. You can use this guide together with the code samples provided with the ToF SDK to get started. When you install the Basler ToF Driver software, you will get the following items:

- a GenApi installation
- the ConsumerImplHelper library
- ToF code samples
- this Programmer's Guide

Installations under Linux

On computers running Linux operating systems, the ToF Driver software is installed by unpacking a .tar.gz archive, either in the default location or a directory of your choice. The C++ samples can then be found in the following location:

```
~/BaslerToF-1.x.y/Samples/Cpp
```

This directory also contains the README.linux file, which explains the build process under Linux.
Code samples for specific tasks are located in subdirectories.
The build process uses a separate makefile for each sample. Therefore, the build process can be started by typing 'make' in the corresponding sample subdirectory.

Installations under Windows

Under Windows, the ToF Driver software is installed using an executable installer, either in the default location or a directory of your choice. This is the default location:

```
"C:\Program Files (x86)\Basler\ToF"
```

The C++ samples are located in the following subdirectory of the installation directory:

```
"\Samples\Cpp"
```

The build process is backed by a Visual Studio solution file named Cpp.sln, which is located in the Samples directory.

Build Process

To build your own application, a straightforward approach under Linux would be to modify the sources of one of the examples according to your requirements. Copy the makefile from the sample folder. Change the values of `NAME` and `SRCS` at the top of the file.

To create your own application under Windows you would generate a new Visual Studio project and copy the project settings from one of the sample projects to your project.

Chapter 2

Getting Started

Overview

This chapter provides a brief overview over the basic steps for connecting to a camera from C++ code, reading and setting camera parameters, starting and stopping data acquisition and processing the acquired data.

GenTL and the ConsumerImplHelper API

On the host computer, you can access ToF cameras via a GenICam-compliant GenTL producer. This GenTL producer is a dynamic library implementing a standardized software interface.

The software interacting with the GenTL producer, like the code samples provided with the ToF SDK, is called a GenTL consumer.

Because it may be a bit cumbersome at times to use the GenTL Producer interface directly, Basler provides the **ConsumerImplHelper** library. This C++ template library serves as an easy to use API providing access to the Basler GenTL producer for ToF cameras. Access to the library is provided mainly by the `CToFCamera` class.

2.1 Initializing the GenTL Producer and Opening a ToF Camera

The first step each program using the Basler ToF GenTL producer has to perform is to load and initialize the producer. This is done by calling the `CToFCamera::InitProducer()` static method.

Afterwards, an instance of the `CToFCamera` class can be created. By creating it, this instance is not yet connected to a physical device. To enumerate all available camera devices and to open the first one found, use the `Open↩FirstCamera()` method. That way, a network connection to the device will be established. The `CToFCamera` class instance uses this connection for communicating with the camera and grabbing depth and image data.

Example (abbreviated):

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    try
    {
        // initialize ToF Genicam Producer
        CToFCamera::InitProducer();

        // ToFcamera convenience class
        CToFCamera camera;

        // Open the first camera found.
        camera.OpenFirstCamera();
        ...
        // if everything went well ..
        exitCode = EXIT_SUCCESS;
    } catch ( ... ) {}
    return exitCode;
}
```

2.2 Accessing Camera Parameters

The `CToFCamera` class provides an API for accessing camera parameters, as shown in the following code sample.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    try
    {
        CToFCamera::InitProducer();
        CToFCamera camera;
        camera.OpenFirstCamera();
        ...
        // Get the integer nodes describing the camera's reggion of interest (ROI).
        CIntegerPtr ptrOffsetX( camera.GetParameter("OffsetX"));
        CIntegerPtr ptrOffsetY( camera.GetParameter("OffsetY"));
        CIntegerPtr ptrWidth( camera.GetParameter("Width"));
        CIntegerPtr ptrHeight( camera.GetParameter("Height"));

        // set ROI to 240x320 pixel
        ptrWidth->SetValue(240);
        ptrHeight->SetValue(320);

        // close camera
        camera.close();

        exitCode = EXIT_SUCCESS;
    } catch ( ... ) {}
    return exitCode;
}
```

Accessing camera parameters using the ConsumerImplHelper API and GenApi will be explained in more detail in the chapter about GenAPI.

2.3 Retrieving Depth Data and Images

The `CToFCamera` class provides the following convenience method:

`GrabContinuous(size_t nBuffers, uint32_t timeout_ms, C* pClass, *callback)`

This method provides the easiest way to continuously grab data from a ToF camera.

Parameters:

- **size_t nBuffers**
  Depth and image data are stored in buffers, where nBuffers is the number of buffers to be used by the method.
- **uint32_t timeout_ms**
  Specifies the amount of time that the method will wait for the next image.
- **C∗ pClass**
  Reference to the object providing a callback method.
- **C::∗onImageAvailable**
  Pointer to a member function that will be called once an image is available.

Signature of the callback method:

```
bool onImageAvailable (GrabResult, BufferParts)
```

Parameters:

- **GrabResult**
  GrabResult struct, containing information about the grabbed data.
- **BufferParts**
  Vector of PartInfo structs. The structs contain a pointer to the actual depth and image data as well as image width and height.

Return Value

- **bool**
  As long as the method returns `true`, GrabContinuous will continue to grab. Once the callback returns `false`, image acquisition will be stopped.

The code sample shows how grabbing can be started immediately after configuring the camera.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    try
    {
        CToFCamera::InitProducer();
        CToFCamera camera;
        camera.OpenFirstCamera();
        ...
        // Get access to a camera parameter.
        CIntegerPtr ptrDepthMax( camera.GetParameter("DepthMax"));

        // Set a parameter value.
        ptrWidth->SetValue(2000);

        // Set some more parameters, as required by the application ...
        ...

        // Acquire images until the call-back onImageGrabbed indicates to stop acquisition.
        // 5 buffers are used (round-robin).
        m_Camera.GrabContinuous( 5, 1000, this, &Sample::onImageGrabbed );

        // close camera
        camera.close();

        exitCode = EXIT_SUCCESS;
    } catch ( ... ) {}
    return exitCode;
}
```

2.4 Processing Image Data

The following method can be used to do some processing on the image data.

```
onImageGrabbed(GrabResult, BufferParts)
```

For an initial check of the validity of the data, the `GrabResult` struct provides the following information (amongst others):

- **frameID**
  The frame ID of the image received, as set by the camera.
- **status**
  The status may be one of the following:
  - `Ok`: The grab was successful and the image data is valid.
  - `Failed`: The grab failed and any attached image data is invalid.
  - `Timeout`: A timeout occurred while waiting for the next image.

The status should be checked prior to accessing the image data.

Depending on the camera configuration, each grabbed frame may consist of multiple parts, each part containing either depth, confidence, or intensity data. `BufferParts` is a vector of one or more `PartInfo` structs. Each `PartInfo` contains a pointer to the actual data of the part of a frame as well as the image width and height, amongst others.

The following code sample shows an example of a callback implementation.

Example:

```
bool Sample::onImageGrabbed( GrabResult grabResult, BufferParts parts )
{
    if ( grabResult.status == GrabResult::Timeout )
    {
        cerr << "Timeout occurred. Acquisition stopped." << endl;
        return false; // Indicate to stop acquisition
    }
    m_nBuffersGrabbed++;
    if ( grabResult.status != GrabResult::Ok )
    {
        cerr << "Image " << m_nBuffersGrabbed << "was not grabbed." << endl;
    }
    else
    {
        // Retrieve the values for the center pixel
        const int width = (int) parts[0].width;
        const int height = (int) parts[0].height;
        const int x = (int) (0.5 * width);
        const int y = (int) (0.5 * height);
        CToFCamera::Coord3D *p3DCoordinate = (CToFCamera::Coord3D*) parts[0].pData + y * width + x;
        uint16_t *pIntensity = (uint16_t*) parts[1].pData + y * width + x;
        uint16_t *pConfidence = (uint16_t*) parts[2].pData + y * width + x;

        cout << "Center pixel of image " << setw(2) << m_nBuffersGrabbed << ": ";
    }
    return m_nBuffersGrabbed < 10; // Indicate to stop acquisition when 10 buffers are grabbed
}
```

Chapter 3

Use cases

In this chapter, several uses cases will be discussed. For each use case, an overview over the key concepts is given at the beginning.

To further illustrate the key concepts, (comprehensively commented) excerpts from the code samples that come with the ToF SDK are included.

3.1 Basic Use Case

3.1.1 Overview

In this use case, as in all the following, the GenICam GenApi library is used to access the camera parameters.

It is shown how to set up the camera to enable 3D (point cloud) data, intensity data, and confidence data.

Excerpts from FirstSample code sample are used to illustrate how to grab images from a ToF camera and how to access the image and depth data.

The GenApi sample, which is discussed in the following chapter, illustrates in more detail how to configure a camera using the GenICam API.

3.1.2 Main Method

The code sample shows how to use the main method to configure the camera in an efficient way, start image acquisition, and release the GenTL producer and all of its resources before terminating the program.

Using the `CToFCamera` class provided by the ConsumerImplHelper API, the proper GenTL Producer for Basler ToF cameras can be initialized by calling the static `CToFCamera::InitProducer()` function.

The `Sample` class is instantiated and its `run()` method is called.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_SUCCESS;

    try
    {
        // Initialize GenTL producer
        CToFCamera::InitProducer();

        // Instantiate Sample class
        Sample processing;

        // Start the run() method of the Sample class
        exitCode = processing.run();
    }
    catch ( GenICam::GenericException& e )
    {
        // Catch exceptions
        cerr << "Exception occurred: " << endl << e.GetDescription() << endl;
        exitCode = EXIT_FAILURE;
    }

    // Release the GenTL producer and all of its resources.
    // Note: Don't call TerminateProducer() until the destructor of the CToFCamera
    // class has been called. The destructor may require resources which may not
    // be available anymore after TerminateProducer() has been called.
    if ( CToFCamera::IsProducerInitialized() )
        CToFCamera::TerminateProducer();  // Won't throw any exceptions

    cout << endl << "Press Enter to exit." << endl;
    while (cin.get() != '\n');

    return exitCode;
}
```

3.1.3 Enabling 3D, Intensity, and Confidence Data

On the host computer, ToF cameras are accessed via the GenICam-compliant Basler GenTL Producer for ToF cameras. The GenTL producer is a dynamic library implementing a standardized software interface.

The GenTL producer provides an acquisition mechanism that has to be run continuously within a grab loop.
The `CToFCamera` can take over this part by providing a grab loop and automated buffer handling, thus effectively freeing the user from having to implement these parts himself.
In this case, the user only needs to provide a callback method.
Within this callback, the user will then be able to access, and process, the image data.

To configure the ToF camera, the GenAPI is used. This is a library that provides the parameters of a camera in the form of GenAPI nodes.
The `GetParameter()` method of the `CToFCamera` class is used to gain access to these nodes. Once a node object has been accessed, read and write access is possible using the GenAPI. This procedure will be explained in more detail in the section covering the GenAPI use case.

Please see the `GenICam Standard` for a detailed description of the GenAPI.

Example:

```
int Sample::run()
{
    m_nBuffersGrabbed = 0;

    try
    {
        // CToFCamera enumerates available camera and opens first one
        m_Camera.OpenFirstCamera();
        cout << "Connected to camera " << m_Camera.GetCameraInfo().strDisplayName << endl;

        // Enable 3D (point cloud) data, intensity data, and confidence data
        GenApi::CEnumerationPtr ptrImageComponentSelector = m_Camera.GetParameter("ImageComponentSelector")
      ;
        GenApi::CBooleanPtr ptrImageComponentEnable = m_Camera.GetParameter("ImageComponentEnable");
        GenApi::CEnumerationPtr ptrPixelFormat = m_Camera.GetParameter("PixelFormat");

        // Enable range data
        ptrImageComponentSelector->FromString("Range");
        ptrImageComponentEnable->SetValue(true);
        // Range information can be sent either as a 16-bit grey value image or as 3D coordinates (point
       cloud). For this sample, we want to acquire 3D coordinates.
        // Note: To change the format of an image component, the Component Selector must first be set to
       the component
        // you want to configure (see above).
        // To use 16-bit integer depth information, choose "Mono16" instead of "Coord3D_ABC32f".
        ptrPixelFormat->FromString("Coord3D_ABC32f" );

        // Enable intensity data
        ptrImageComponentSelector->FromString("Intensity");
        ptrImageComponentEnable->SetValue(true);

        // Enable confidence data
        ptrImageComponentSelector->FromString("Confidence");
        ptrImageComponentEnable->SetValue(true);

        // Acquire images until the call-back onImageGrabbed indicates to stop acquisition.
        // 5 buffers are used (round-robin).
        m_Camera.GrabContinuous( 5, 1000, this, &Sample::onImageGrabbed );

        // Clean-up
        m_Camera.Close();
    }
    catch ( const GenICam::GenericException& e )
    {
        cerr << "Exception occurred: " << e.GetDescription() << endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Because it may be a bit cumbersome at times to use the GenTL Producer API directly, Basler provides the **ConsumerImplHelper** API, which takes care of finding and initializing the GenTL producer for ToF cameras.

Once the GenTL producer has been initialized (see main() method), an instance of the `CToFCamera` can be created. After creating the instance, it must be connected to a physical camera device. This can be achieved by using the `OpenFirstCamera()` method. This method enumerates all attached cameras and establishes a connection to the first one found.

3.1.4 Processing Intensity Data

The `CToFCamera` class also provides an easy way for accessing grabbed image data, as shown in the following code sample.

Example:

```
bool Sample::onImageGrabbed( GrabResult grabResult, BufferParts parts )
{
    if ( grabResult.status == GrabResult::Timeout )
```

```
    {
        cerr << "Timeout occurred. Acquisition stopped." << endl;
        return false; // Indicate to stop acquisition
    }
    m_nBuffersGrabbed++;
    if ( grabResult.status != GrabResult::Ok )
    {
        cerr << "Image " << m_nBuffersGrabbed << "was not grabbed." << endl;
    }
    else
    {
        // Retrieve the values for the center pixel
        const int width = (int) parts[0].width;
        const int height = (int) parts[0].height;
        const int x = (int) (0.5 * width);
        const int y = (int) (0.5 * height);
        CToFCamera::Coord3D *p3DCoordinate = (CToFCamera::Coord3D*) parts[0].pData + y * width + x;
        uint16_t *pIntensity = (uint16_t*) parts[1].pData + y * width + x;
        uint16_t *pConfidence = (uint16_t*) parts[2].pData + y * width + x;

        cout << "Center pixel of image " << setw(2) << m_nBuffersGrabbed << ": ";
        cout.setf( ios_base::fixed);
        cout.precision(1);
        if ( p3DCoordinate->IsValid() )
            cout << "x=" << setw(6) <<  p3DCoordinate->x << " y=" << setw(6) << p3DCoordinate->y << " z=" <
      < setw(6) << p3DCoordinate->z;
        else
            cout << "x=   n/a y=   n/a z=   n/a";
        cout << " intensity="<< setw(5) << *pIntensity << " confidence=" << setw(5) << *pConfidence << endl
      ;
    }
    return m_nBuffersGrabbed < 10; // Indicate to stop acquisition when 10 buffers are grabbed
}
```

How to access GenAPI nodes using the ConsumerImplHelper API will be explained in more detail in the following chapter.

3.2 Configuring a Camera Using the GenICam API

3.2.1 Overview

This chapter explains how to use the GenICam API for reading and writing parameters of the ToF camera. Excerpts from the GenAPI code sample are used to explain the key concepts.

3.2.2 Parameter Access via Pointers

The smart pointers of the GenApi are used as a convenient way to access parameters.

Smart Pointers

A smart pointer in C++ is a class which wraps a 'raw' (or 'bare') C++ pointer in order to manage the lifetime of the object being pointed to. Various different implementations are available, e.g., provided by the Boost library or by C++11.

The use of smart pointers mainly prevents the user from forgetting to delete them after use, which would generate a memory leak. Therefore, they should be preferred over raw pointers.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    try
    {
        // Initialise ToF producer, which presents the Genicam interface
        CToFCamera::InitProducer();

        // Create a ToF camera object
        CToFCamera camera;

        // Open the camera
        camera.OpenFirstCamera();

        // Get access to a parameter named "SomeParameter"
        CIntegerPtr ptrFancyParameter = camera.GetParameter( "SomeParameter" );

        ...

        // Close camera
        camera.Close();

        exitCode = EXIT_SUCCESS;

    } catch ( GenICam::GenericException& e )
    {
        cerr << "Exception occurred: " << endl << e.GetDescription() << endl;
    }

    ...

    return exitCode;
}
```

In order to understand how camera parameters are accessed by a GenTL consumer, it is useful to know that they are organized and accessed as so-called "nodes" in the GenAPI reference implementation.

The GenICam specification defines a format for camera description files.
These files describe the configuration interface of GenICam-compliant cameras. The description files are written in XML and describe camera registers, their interdependencies, and all other information needed to access high-level features such as Gain, Exposure Time, or Image Format by means of low-level register read and write operations.

The elements of a camera description file are represented as software objects called "nodes". For example, a node can represent a single camera register, a camera parameter such as Gain, a set of available parameter values, etc. Each node implements the `GenApi::INode` interface.

The nodes are linked by different relationships as explained in the GenICam standard document available at GenICam.org. The complete set of nodes is stored in a data structure called a "node map".
At runtime, the `CToFCamera` class instantiates a node map from an XML description provided by the GenTL producer. The `CToFCamera::GetParameter()` method is used to get access to a GenApi node that provides access to a camera parameter.

The names and types of the parameter nodes can be found by using the documentation panes of the ToF Viewer or the pylon Viewer.

The GenICam GenApi standard document can be downloaded here.
The GenTL standard document can be downloaded from here.

3.2.3 Smart Pointer Types

GenApi supports various different parameter types. For each of them, a corresponding smart pointer is available. The following smart pointer types are used within the samples code:

- **CIntgegerPtr**
  Used for integer nodes.
- **CFloatPtr**
  Used for nodes representing float values.
- **CEnumerationPtr**
  Used for enumeration nodes.
- **CBooleanPtr**
  Used for boolean nodes.

Instances of these pointer classes are created by assigning them the return value of the `CToFCamera::GetParameter()` method.

3.2.4 Validity Check

Even if the camera doesn't provide a requested parameter, the camera class returns a smart pointer. If the user tried to access the smart pointer, an exception will be thrown since there is no corresponding GenApi node object that the pointer could provide access to.
Therefore, a check is necessary to see whether the result returned by the `GetParameter` method is a valid pointer to a parameter.

3.2.5 Checking Accessibility of Nodes

Parameters may be read-only or (temporarily) not available, depending on the current state of the camera device. When accessing parameters that are not available or trying to change a read-only parameter, an exception will be thrown. The `GenApi::IsWritable()` function can be used to check whether a parameter is available and a value can be set.
GenApi provides some further convenience functions for checking the accessibility of nodes: `GenApi::IsReadable()`, `GenApi::IsAvailable()`, and `GenApi::IsImplemended()`. These functions work with all parameter types.

Parameters that don't exist can't be read.

The following excerpt from the GenAPI sample accesses several GenApi nodes while applying additional checks to assure validity and accessibility of the parameters. Enumeration parameters will be discussed further down.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    try
    {
        CToFCamera::InitProducer();
        CToFCamera camera;
        camera.OpenFirstCamera();

        CIntegerPtr ptrFancyParameter = camera.GetParameter( "FancyParameter" );

        // Check to see whether GetParameter returned a valid pointer to a parameter:
        if ( ! ptrFancyParameter.IsValid() )
        {
            cout << "Camera doesn't provide the parameter 'FancyParameter'" << endl;
        }

        // Check readability of parameter
        if ( ! IsReadable(ptrFancyParameter) )
        {
            cout << "'FancyParameter' is not readable."  << endl;
        }

        CFloatPtr ptrDeviceTemperature = camera.GetParameter ( "DeviceTemperature" );

        // Check writeability of parameter
        if ( ! IsWritable(ptrDeviceTemperature) )
        {
            cout << "Device temperature is not writable." << endl;
        }

        // All parameter types can be converted to a string using the ToString() method.
        if ( IsReadable(ptrDeviceTemperature) )
        {
            cout << "Current temperature is " << ptrDeviceTemperature->ToString() << endl;
        }

        // Get the integer nodes describing the camera's area of interest (AOI).
        CIntegerPtr ptrOffsetX( camera.GetParameter("OffsetX"));
        CIntegerPtr ptrOffsetY( camera.GetParameter("OffsetY"));
        CIntegerPtr ptrWidth( camera.GetParameter("Width"));
        CIntegerPtr ptrHeight( camera.GetParameter("Height"));

        // Set the AOIs offset (i.e. upper left corner of the AOI) to the allowed minimum.
        ptrOffsetX->SetValue( ptrOffsetX->GetMin() );
        ptrOffsetY->SetValue( ptrOffsetY->GetMin() );

        // Some properties have restrictions, i.e., the values must be a multiple of a certain
        // increment. Use GetInc/GetMin/GetMax to make sure you set a valid value, as shown
        // in the Adjust() function above.
        int64_t newWidth = 202;
        newWidth = Adjust(newWidth, ptrWidth->GetMin(), ptrWidth->GetMax(), ptrWidth->GetInc());

        int64_t newHeight = 101;
        newHeight = Adjust(newHeight, ptrHeight->GetMin(), ptrHeight->GetMax(), ptrHeight->GetInc());

        ptrWidth->SetValue(newWidth);
        ptrHeight->SetValue(newHeight);

        cout << "OffsetX          : " << ptrOffsetX->GetValue() << endl;
        cout << "OffsetY          : " << ptrOffsetY->GetValue() << endl;
        cout << "Width            : " << ptrWidth->GetValue() << endl;
        cout << "Height           : " << ptrHeight->GetValue() << endl;

        // Reset the AOI to its full size.
        ptrWidth->SetValue(ptrWidth->GetMax() );
        ptrHeight->SetValue(ptrHeight->GetMax());

        CFloatPtr ptrAcquisitionFrameRate( camera.GetParameter("AcquisitionFrameRate") );
        cout << "Current acquisition frame rate: " << ptrAcquisitionFrameRate->GetValue();
        // Set to maximum
        ptrAcquisitionFrameRate->SetValue( ptrAcquisitionFrameRate->GetMax() );
        cout << ". New value: " << ptrAcquisitionFrameRate->GetValue() << endl;

        //
        // Configuring the camera's exposure time
        //
        //
        // By default, the exposure time is controlled automatically. Switch off the
        // automatic control first.
        CEnumerationPtr( camera.GetParameter("ExposureAuto"))->FromString("Off");
        // Now the value can be controlled manually.
        CFloatPtr ptrExposureTime( camera.GetParameter("ExposureTime") );
```

```
        ptrExposureTime->SetValue( ptrExposureTime->GetMin() * 3 );
        cout << "Set exposure time to " << ptrExposureTime->GetValue() << endl;

        camera.Close();
        exitCode = EXIT_SUCCESS;
    }
    catch ( GenICam::GenericException& e )
    {
        cerr << "Exception occurred: " << endl << e.GetDescription() << endl;
    }

    // Release the GenTL producer and all of its resources.
    // Note: Don't call TerminateProducer() until the destructor of the CToFCamera
    // class has been called. The destructor may require resources which may not
    // be available anymore after TerminateProducer() has been called.
    if ( CToFCamera::IsProducerInitialized() )
        CToFCamera::TerminateProducer();  // Won't throw any exceptions

    cout << endl << "Press Enter to exit." << endl;
    while (cin.get() != '\n');

    return exitCode;
}
```

3.2.6 Enumeration Nodes, Selectors, and Boolean Parameters

The use of enumeration parameters, selectors, and Boolean parameters deserves special attention.

Enumeration parameters, e.g., PixelFormat, represent a defined set of named values.

Selectors are enumeration parameters. They are used to select the parameter that the user wants to configure.

For example, the ComponentSelector parameter is a selector. With this parameter the user selects the part of an acquired image that the user wants to configure. The selected part can then be enabled using the Component↩ Enable parameter. Enabling a part means that the corresponding data will be transmitted by the camera.

For some parts, different pixel formats are available. Using the PixelFormat parameter, the user can choose the desired pixel format for the part selected with the ComponentSelector parameter.

All parameter types provide a corresponding `FromString()` method that allows the user to set a parameter by using a string. The `FromString()` method is the only way to set the value of an enumeration parameter.

The following code sample shows how to iterate over all parts and enable only those parts that support RGB data.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    try
    {
        CToFCamera::InitProducer();
        CToFCamera camera;
        camera.OpenFirstCamera();

        CEnumerationPtr ptrImageComponentSelector = camera.GetParameter("ImageComponentSelector");
        CBooleanPtr ptrImageComponentEnable = camera.GetParameter("ImageComponentEnable");
        CEnumerationPtr ptrPixelFormat( camera.GetParameter("PixelFormat") );
        NodeList_t entries;
        ptrImageComponentSelector->GetEntries( entries );
        for ( NodeList_t::const_iterator entry = entries.begin(); entry != entries.end(); ++entry)
        {
```

```
            CEnumEntryPtr ptrEntry( *entry );
            cout << "Camera supports a " << ptrEntry->GetSymbolic() << " part: " << (IsAvailable( ptrEntry
    ) ? "yes" :  "no") << endl;
            if ( IsAvailable( ptrEntry ) )
            {
                // First, select the part to configure...
                ptrImageComponentSelector->FromString( ptrEntry->GetSymbolic() );
                // ... then access the parameters for the part.
                if ( IsAvailable( ptrPixelFormat->GetEntryByName( "RGB8")))
                {
                    ptrPixelFormat->FromString( "RGB8");
                    ptrImageComponentEnable->SetValue(true);
                }
                cout << "Pixel format of part " << ptrImageComponentSelector->ToString() << ": " <<
    ptrPixelFormat->ToString() << endl;
                cout << "Part is " << (ptrImageComponentEnable->GetValue() ? "enabled" : "disabled") <<
    endl;
            }
        }
                camera.Close();
        exitCode = EXIT_SUCCESS;
    }
    catch ( GenICam::GenericException& e )
    {
        cerr << "Exception occurred: " << endl << e.GetDescription() << endl;
    }

    if ( CToFCamera::IsProducerInitialized() )
        CToFCamera::TerminateProducer();  // Won't throw any exceptions

    cout << endl << "Press Enter to exit." << endl;
    while (cin.get() != '\n');

    return exitCode;
}
```

3.3 Using a Custom Grab Loop

3.3.1 Overview

This chapter explains how to set up a custom grab loop.

It also shows how buffers can be allocated, queued and released and how ownership of the buffers is handled.

Using the CustomGrabLoop code sample, it is shown how the grab result is processed and the image data is extracted.

Also, ways to handle potential timeouts will be shown.

3.3.2 Setting up the Camera

The camera is set up as in the previous use cases, which means that range data, confidence map, and 3D data are enabled using the ComponentSelector and ComponentEnable parameters.

3.3.3 Allocating Buffers

When employing a custom grab loop, the buffer allocator, which provides the image buffers, can either be provided by the `CToFCamera` class, which uses its default buffer allocator, or it can be provided by the application.

The excerpt from the CustomGrabLoop code sample shows how buffers can be allocated using a custom allocator. After calling the `PrepareAcquisition( nBuffers )` method, the camera object creates n buffers. By calling the `m_Camera.QueueBuffer(..)` method for each buffer, ownership over the allocated buffer is given to the camera.

Once the acquisition engine has been started, the camera can start to fill the buffers with image data.

Example (abbreviated):

```
int Sample::run()
{
    const size_t nBuffers = 3;  // Number of buffers to be used for grabbing.
    const size_t nImagesToGrab = 10; // Number of images to grab.
    size_t nImagesGrabbed = 0;

    try
    {
        //
        // Open and parameterize the camera.
        //

        setupCamera();

        //
        // Grab and process images
        //

        // Let the camera class use our allocator.
        // When the application doesn't provide an allocator, a default one that allocates memory buffers
        // on the heap will be used automatically.
        m_Camera.SetBufferAllocator( new CustomAllocator(), true); // m_Camera takes ownership and will
       clean-up allocator.

        // Allocate the memory buffers and prepare image acquisition.
        m_Camera.PrepareAcquisition( nBuffers );

        // Enqueue all buffers to be filled with image data.
        for ( size_t i = 0; i < nBuffers; ++i )
        {
            m_Camera.QueueBuffer( i );
        }

        // Start the acquisition engine.
        m_Camera.StartAcquisition();

        // Now, the acquisition can be started on the camera.
        m_Camera.IssueAcquisitionStartCommand(); // The camera continuously sends data now.

        // ================================================
        // Grab loop omitted for clarity at this point!
        // ================================================

        // Stop the camera
        m_Camera.IssueAcquisitionStopCommand();

        // Stop the acquisition engine and release memory buffers and other resources used for grabbing.
        m_Camera.FinishAcquisition();

        // Close the connection to the camera
        m_Camera.Close();

      } catch ( const GenICam::GenericException& e )
      {
          // ... do some error handling here
          return EXIT_FAILURE;
      }
    return nImagesGrabbed == nImagesToGrab ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

3.3.4 Grab Loop

The grab loop, which has been omitted in the previous code excerpt to keep it short, is repeated until acquisition of the specified number of images has been completed.

Note that, after the image data has been processed in the `processData()` method, the buffer just processed is returned back to the camera by calling the `m_Camera.QueueBuffer()` method. The `processData()` method will be explained in detail further down. Calling the `m_Camera.QueueBuffer()` method, returns the ownership of the buffer to the camera. Once the buffer is returned to the camera, it should not be written to anymore because the camera can now overwrite the buffer with new image data.

```
// ... see previous code excerpt

// Enter the grab loop
do
{
    // Use a grabresult struct to return the image data and acquisition status
    GrabResult grabResult;
    // Wait up to 1000 ms for the next grabbed buffer available in the
    // acquisition engine's output queue.
    m_Camera.GetGrabResult( grabResult, 1000 );

    // Read status of grabresult to check whether a buffer has been grabbed successfully.

    // Handle acqusition timeouts here
    if ( grabResult.status == GrabResult::Timeout )
    {
        cerr << "Timeout occurred." << endl;
        // The timeout might be caused by a removal of the camera. Check if the camera
        // is still connected.
        if ( ! m_Camera.IsConnected() )
        {
            cerr << "Camera has been removed." << endl;
        }
        break; // exit loop
    }

    // handle complete acqusisition failure
    if ( grabResult.status != GrabResult::Ok )
    {
        cerr << "Failed to grab image." << endl;
        break; // exit loop
    }
    nImagesGrabbed++; // count image grabbed

    // We can process the buffer now. The buffer will not be overwritten with new data until
    // it is explicitly placed in the acquisition engine's input queue again.
    processData( grabResult );

    // We finished processing the data, put the buffer back into the acquisition
    // engine's input queue to be filled with new image data.
    m_Camera.QueueBuffer( grabResult.hBuffer );

} while ( nImagesGrabbed < nImagesToGrab );

// ... see previous code excerpt
```

3.3.5 Image Processing

If no timeout or general error has occurred, the following method is called:

```
processData( grabResult )
```

Here, image extraction from the buffers and further processing of image data can be handled.
The following excerpt from the CustomGrabLoop Sample may serve as an example.

Example:

```
void Sample::processData( const GrabResult& grabResult )
{
    // Vector holding PartInfo structs
    BufferParts parts;

    // retrieve the BufferParts vector from the grab result
    m_Camera.GetBufferParts( grabResult, parts);

    // Retrieve the values for the center pixel
    const int width = (int) parts[0].width;
    const int height = (int) parts[0].height;
    const int x = (int) (0.5 * width);
    const int y = (int) (0.5 * height);
    CToFCamera::Coord3D *p3DCoordinate = (CToFCamera::Coord3D*) parts[0].pData + y * width + x;
    uint16_t *pIntensity = (uint16_t*) parts[1].pData + y * width + x;
    uint16_t *pConfidence = (uint16_t*) parts[2].pData + y * width + x;
```

```
    cout.setf( ios_base::fixed);
    cout.precision(1);
    if ( p3DCoordinate->IsValid() )
        cout << "x=" << setw(6) <<  p3DCoordinate->x << " y=" << setw(6) << p3DCoordinate->y << " z=" <<
      setw(6) << p3DCoordinate->z;
    else
        cout << "x=   n/a y=   n/a z=   n/a";
    cout << " intensity="<< setw(5) << *pIntensity << " confidence=" << setw(5) << *pConfidence << endl;
}
```

3.4 Parametrize Camera from a File

3.4.1 Overview

This chapter explains how to configure the camera from a pylon feature stream file (.pfs). .pfs files can be generated using the ToF Viewer (Windows) or the pylon Viewer (Linux).
Code excerpts from the ParametrizeFromFile code sample are used to explain the implementation.

3.4.2 Reading the .pfs File

The `CToFCamera` class contains the following method for reading a .pfs file and parameterizing the camera:

```
ParametrizeFromFile(const char* parameterFileName)
```

Example:

```
int main(int argc, char* argv[])
{
    CToFCamera camera;

    int exitCode = EXIT_FAILURE;
    if ( argc != 2 )
    {
        printUsage( argv );
        goto exit;
    }

    try
    {
        CToFCamera::InitProducer();
        camera.OpenFirstCamera();
        camera.ParametrizeFromFile( argv[1] );

        // Work with camera...

        camera.Close();
    }
    catch ( GenICam::GenericException& e )
    {
        cerr << "Exception occurred: " << endl << e.GetDescription() << endl;
        // After successfully opening the camera, the IsConnected method can be used
        // to check if the device is still connected.
        if ( camera.IsOpen() && ! camera.IsConnected() )
        {
            cerr << "Camera has been removed." << endl;
        }
    }

    // Release the GenTL producer and all of its resources.
    // Note: Don't call TerminateProducer() until the destructor of the CToFCamera
    // class has been called. The destructor may require resources which may not
    // be available anymore after TerminateProducer() has been called.
    if ( CToFCamera::IsProducerInitialized() )
        CToFCamera::TerminateProducer();  // Won't throw any exceptions
    exitCode = EXIT_SUCCESS;

exit:
    cout << endl << "Press Enter to exit." << endl;
    while (cin.get() != '\n');

    return exitCode;
}
```

3.5 Saving Point Cloud Data

3.5.1 Overview

This chapter explains how to save 3D data to a plain text file in the .pcd format.

The .pcd file format is the file format used by the Point Cloud Library. It will be explained here and an example of a .pcd file will be given.

At the end, a walk-through of the code sample will be given.

3.5.2 The Point Cloud Library Data Format

Point cloud data is stored by the Point Cloud Library in a data format defined here: `.pcd file format`.

The file starts with a header, which is encoded in ASCII and has the following fields:

PCL Header

Field	Meaning
VERSION	Version of the .pcd file format definition
FIELDS	Content of one data field, example: x y z rgb # XYZ + colors
SIZE	Specifies the size of each dimension in bytes
TYPE	Number type, I=integer, U=unsigned integer, F=float
COUNT	Number of elements per dimension
WIDTH	Image width
HEIGHT	Image height
VIEWPOINT	Can be used to specify the coordinates of the camera
POINTS	Total number of points (= width $*$ height for one image)
DATA	Data type that the point cloud data is stored in

Data

After the DATA field of the header, the data itself starts.
The data can be in one of two formats.

- In human-readable form as ASCII data. In this case each line of the file will contain the data of one point.
- As a binary blob.

Example of the Contents of a .PCD File

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
0.81915 0.32 0 4.2108e+06
...
...
```

3.5.3 Code Sample, Part 1

In the main method of the code sample, firstly the camera is set up to acquire 3D and intensity data.

Example:

```
int main(int argc, char* argv[])
{
    int exitCode = EXIT_FAILURE;

    // The name of the .pcd file to save the data to
    const char* fileName = argc > 1 ? argv[1] : "points.pcd";

    CToFCamera camera;
    try
    {
        // Initialize producer and open a camera
        CToFCamera::InitProducer();
        camera.OpenFirstCamera();

        // Start selecting image components to be retrieve
        CEnumerationPtr ptrComponentSelector = camera.GetParameter("ComponentSelector");

        CBooleanPtr ptrComponentEnable = camera.GetParameter("ComponentEnable");
        CEnumerationPtr ptrPixelFormat = camera.GetParameter("PixelFormat");

        /* Parameterize the camera to send 3D coordinates       */

        // Start with the "Range" component
        ptrComponentSelector->FromString("Range");

        // Enable range data
        ptrComponentEnable->SetValue( true );

        // Set pixelformat to "Coord3D_ABC32f" fo rthe range data
        ptrPixelFormat->FromString("Coord3D_ABC32f");

        // Select intensity data
        ptrComponentSelector->FromString("Intensity");

        // Enable intensity data
        ptrComponentEnable->SetValue(true);

        // Select "Mono16" as format of the intensity data
        ptrPixelFormat->FromString("Mono16");

        /* Acquire one single Image                             */

        // Provide bufferParts struct which the grab loop will use to store image data
        BufferParts parts;
```

```
        // Grab one single image
        GrabResultPtr ptrGrabResult = camera.GrabSingleImage( 1000, &parts );

        /* Save the 3D data                                          */

        // Before processing the data, check the status of the grab result
        if ( ptrGrabResult->status == GrabResult::Ok )
        {
            // Call the method where point cloud data will be saved
            SavePointCloud( parts, fileName );
        }
        else
        {
            // Handle possible error during image acquisition here ..
            cerr << "Failed to grab an image." << endl;
        }

        // Close camera and exit
        camera.Close();
        exitCode = EXIT_SUCCESS;
    }
    catch ( GenICam::GenericException& e )
    {

        // Perform error handling here

        cerr << "Exception occurred: " << endl << e.GetDescription() << endl;
        // After successfully opening the camera, the IsConnected method can be used
        // to check if the device is still connected.
        if ( camera.IsOpen() && ! camera.IsConnected() )
        {
            cerr << "Camera has been removed." << endl;
        }

    }

    if ( CToFCamera::IsProducerInitialized() )
        CToFCamera::TerminateProducer();  // Won't throw any exceptions

    cout << endl << "Press Enter to exit." << endl;
    while (cin.get() != '\n');

    return exitCode;
}
```

One single image is acquired and 3D and intensity data are stored in the `BufferParts` struct.

Once the image has been acquired and data has been stored, the `SavePointCloud( parts, fileName )` method is called, which takes the `BufferParts` struct containing the data and the file name as arguments.

3.5.4 Code Sample, Part 2

The `SavePointCloud(..)` method in the code sample shows how storage of data in the .pcd format can be implemented.

Since the `BufferParts` vector is used to store image data, it will be explained here briefly as well. `BufferParts` is a vector holding `PartInfo` structs.

Coord3D is a struct holding the 3D coordinates plus a flag, which can be interrogated to check validity of the data.

```
typedef std::vector<PartInfo>     BufferParts;
```

PartInfo, BufferpartType and Coord3D

```
struct PartInfo
{
    void*          pData;       // the data pointer
    size_t         size;        // size of the data array
    size_t         width;       // width of the image
    size_t         height;      // height of the image
    BufferPartType partType;    // type of buffer data
    uint64_t       dataFormat;  // data format according to Genicam standard
};

enum BufferPartType
{
    Range,                      // Range data
    Intensity,                  // Intensity data
    Confidence,                 // Confidence data
    Undefined                   // type of data is not defined
};

struct Coord3D
{
    float x;                    // X-value
    float y;                    // Y-value
    float z;                    // Z-value
    bool IsValid() const { return z == z; }  // check for NAN
};
```

The `WritePcdHeader(..)` method is used to write the header according to the .pcd data format as explained above.

The method produces the fields requested by the standard. For more information, please refer to the standard.

Example:

```
bool SavePointCloud( const BufferParts& parts, const char* fileName )
{
    // Check wether there is valid data in the BufferParts struct
    if ( parts.empty() )
    {
        cerr << "No valid image data." << endl;
        return false;
    }

    // If the point cloud data format is enabled, the first part always contains the point cloud data.
    if ( parts[0].dataFormat != PFNC_Coord3D_ABC32f )
    {
        cerr << "Unexpected data format for the first image part. Coord3D_ABC32f is expected." << endl;
        return false;
    }

    // If available, intensity data should be contained in the second part
    const bool saveIntensity = parts.size() > 1;
    if ( saveIntensity && parts[1].dataFormat != PFNC_Mono16 )
    {
        cerr << "Unexpected data format for the second image part. Mono 16 is expected." << endl;
        return false;
    }

    // Try to create a output file stream with the file
    ofstream o( fileName );
    if ( ! o )
    {
        cerr << "Error:\tFailed to create file "<< fileName << endl;
        return false;
    }

    cout << "Writing point cloud to file " << fileName << "...";

    // Get the 3D data and store it as a Coord3Dstruct
    CToFCamera::Coord3D *pPoint = (CToFCamera::Coord3D*) parts[0].pData;

    // save intensity data, if available
    uint16_t *pIntensity = saveIntensity ? (uint16_t*) parts[1].pData : NULL;
```

```
    // calculate number of pixels from width/height of image
    const size_t nPixel = parts[0].width * parts[0].height;

    // Write the header of the .PCL file into the output file stream
    WritePcdHeader( o, parts[0].width, parts[0].height, saveIntensity );

    // iterate over all pixels
    for ( size_t i = 0; i < nPixel; ++i )
    {
        // Check if there are valid 3D coordinates for that pixel.
        if ( pPoint->IsValid() )
        {
            // set precision of output file stream
            o.precision( 0 );  // Coordinates will be written as whole numbers.

            // Write the coordinates of the next point. Note: Since the coordinate system
            // used by the CloudCompare tool is different from the one used by the ToF camera,
            // we apply a 180-degree rotation around the x-axis by writing the negative
            // values of the y and z coordinates.
            o << std::fixed << pPoint->x << ' ' << -pPoint->y << ' ' << -pPoint->z;

            if ( saveIntensity )
            {
                // Save the intensity as an RGB value.
                uint8_t gray = *pIntensity >> 8;
                uint32_t rgb = (uint32_t) gray << 16 | (uint32_t) gray << 8 | (uint32_t) gray;
                // The point cloud library data format represents RGB values as floats.
                float fRgb = * (float*) &rgb;
                o.unsetf(ios_base::floatfield); // Switch to default float formatting
                o.precision(9); // Intensity information will be written with highest precision.
                o << ' ' << fRgb << endl;
            }
        }
        else
        {
            o << "nan nan nan 0" << endl;
        }
        pPoint++;
        pIntensity++;
    }
    o.close();
    cout << "done." << endl;
    return true;
}
```

Viewing Point Cloud Data

Files in the .pcd format can be viewed using the `free CloudCompare software.`

3.6 Multiple Cameras

3.6.1 Overview

In the preceding section the `CToFCamera::OpenFirstCamera()` method was used for establishing a connection to the physical ToF camera device. This method is a convenience function that enumerates all connected ToF cameras and opens the first one found. This is useful if only one camera is connected or if you don't care which camera gets opened. However, if there are multiple cameras connected and you want to open a specific one, the `OpenFirstCamera()` method is not suitable. To choose a specific camera device, you can pass a chosen property of the camera to be opened, e.g., the IP address or the serial number. Alternatively, you can retrieve a list of connected cameras. The application can pick any item from that list and establish a connection to the corresponding camera.

These two use cases are illustrated in the following sections.

Note

If you want to grab data from multiple cameras simultaneously, you have to be aware that the light pulses of multiple cameras capturing the same scene will be interfering with each other. Either assign different device channels to each camera to minimize the interference or use the synchronous fee run feature to avoid multiple cameras sending out light at the same time. Refer to the ToF User's Manual for details about device channels and the synchronized free run feature. The synchronized free run feature is also illustrated in this guide. Refer to the Synchronizing Image Acqusition of Multiple ToF Cameras chapter for code samples.

3.6.2 Opening a Camera by Specifying one of its Properties

A convenient way to select a specific camera is using the `CToFCamera::OpenCamera(CameraInfoKey key, string value)` method. This method performs the following actions:

- enumerates all connected cameras
- picks from that list the first `CameraInfo` whose property specified by `key` is equal to `value`
- passes the picked `CameraInfo` to the `CToFCamera::OpenCamera(CameraInfo)` method
- throws an exception if there is no matching camera

Example:

```
CToFCamera camera;
camera.Open(IpAddress, "192.168.0.2");
```

Instead of the `IpAddress` property, any other property of the `CameraInfo` struct can be used for selecting a specific device, e.g., the `SerialNumber` or the `UserDefinedNames` properties.

Example:

```
CToFCamera camera;
camera.Open(SerialNumber, "237653");
```

3.6.3 Enumerating and Selecting Cameras

To enumerate all available ToF cameras, use the following static function:

```
CToFCamera::EnumerateCameras()
```

`EnumerateCameras()` returns a `CameraList`, which basically is a `std::list`. For each connected camera, the returned list contains one `CameraInfo` element, holding the information about that camera. `CameraInfo` has properties like the camera's IP address, its serial number, and a unique DeviceID. A specific camera device can be opened by picking the desired `CameraInfo` element from the list and passing it to the `CToFCamera::Open(CameraInfo)` method.

Example:

```
// Pseudo code for selecting one or multiple cameras that meet certain criteria.
// For each camera to be opened an instance of the 'CToFCamera' class is created, opened, and stored
// in a vector.
std::vector< std::shared_ptr<CToFCamera> > cameras;
CameraList cameraList = CToFCamera::EnumerateCameras();
for ( CameraList::const_iterator it = cameraList.begin(); it != cameraList.end(); ++it)
{
    if ( *it meets certain criteria )
    {
        std::shared_ptr<CToFCamera> ptrCamera(new CToFCamera);
        ptrCamera->Open(*it);
        cameras.push_back( ptrCamera );
    }
}
```

3.7 Synchronizing Image Acqusition of Multiple ToF Cameras

3.7.1 Overview

This chapter explains how to configure two or more ToF cameras to acquire images synchronously using the synchronous free run feature.

When pointing two ToF cameras at the same target, the cameras will mutually disturb each other with the light patterns they are sending out. The reason is that their respective light patterns are identical. Therefore, light sent out by one camera will also be registered by the other cameras in the setup. The result of this would be inaccurate measurement data.

One way of mitigating the negative effects caused by this mutual disturbance of ToF cameras is to use the multichannel feature. However, this leads to a decrease in the signal-to-noise ratio.

Another method is to synchronize the cameras' internal clocks and then to let them acquire images one after other. This can be achieved by using the synchronous free run feature and offers a superior signal-to-noise ratio.

The basic concepts of synchronous free run as well as the principles of the underlying PTP (Precision Time Protocol) will be explained in the following sections.

3.7.2 About Synchronous Free Run

A camera can be operated in triggered mode or free running. Triggered means that the start of each image exposure is triggered either by an internally or an externally generated signal.
Free running, on the other hand, means, that the sensor of the camera acquires images at its own internal speed.
In synchronous free run, image exposure is triggered by an internal timer. The timer operates based on the intrinsic precision clock of the camera. The clocks of multiple cameras are synchronized by the precision time protocol (PTP).

(In that sense, the notion of 'free run' is slightly misleading. It rather refers to the fact that the trigger signals of the cameras are not generated by a common external trigger source but internally by the cameras themselves. In that sense, in PTP mode, each single camera may be considered free-running.)

3.7.3 About PTP

PTP is a method to synchronize clocks of devices that are connected via Ethernet. It follows a simplified description of the PTP protocol.

(For details, please refer to 1 or 2.)

Amongst devices synchronizing via PTP, one of the devices takes the role of the master clock, while the others are slaves.

So prior to synchronisation, the devices need to negotiate the master role. This is done by means of the Best Master Clock algorithm.

After the master device has been determined, it (resp. the clock instance within the master) begins to send synchronization messages to the slaves so they can synchronize their clocks.
In return, the slaves are sending delay request messages, which the master responds to with delay response messages. From this, the slaves are able to calculate the network delay which has to be added to the timestamps from the master.

Since network delay for example is not constant and not equal in both directions, the PTP algorithm can only estimate the correct time by means of a step-by-step approximation.

Thus, clock synchronization can't be achieved immediately, but improves over time.

3.7.4 Using Synchronous Free Run with Time of Flight cameras

Using PTP, camera clocks can be synchronized with an accuracy of 1 µs or even better. It is therefore possible to operate cameras in two ways:

1. Cameras acquire images exactly at the same point of time
2. Cameras acquire images synchronously but with an intentional delay added between exposures

Scenario 1 is useful when operating normal "2D" cameras. When ToF cameras are running exactly in sync, however, they can disturb each other's measurement accuracy because of the light each camera sends out. Therefore, scenario 2 is preferable. Here, cameras can acquire images using a round-robin approach:

3.7.4.1 Synchronous Free Run in Standard (No HDR) Mode

Let's assume that 4 ToF cameras have been set up to operate synchronously. The processing mode has been set to 'Standard' (no HDR) and exposure time is set to 20 ms.
Camera 1 starts exposure. After 20 ms it finishes exposure (and also disables its infrared LEDs).
Camera 1 starts readout, which will take another 21 ms. However, since the LEDs of camera 1 are already disabled, camera 2 can be set up to start exposure after a delay of 21 ms. Camera 3 starts after 42 ms, camera 4 after 63 ms.
After 84 ms camera 4 has finished exposure. Note that there is an additional 1 ms in order to avoid disturbance between cameras, even taking into account clock jitter or other inaccuracies in the trigger timing.

The trigger rate of all cameras can safely be set to 10 fps, which results in a break of 20 ms, before, after 100 ms, camera 1 once again starts exposure. In this way, the four cameras can be operated synchronously while at the same time avoiding mutual disturbance by delaying exposures.

3.7.4.2 Synchronous Free Run in HDR Mode

HDR is achieved by means of two separate exposures.

Currently, the default values of the Basler ToF Camera are 4 ms for the first exposure time and 20 ms for the second one.
After first and second exposure, there are readout times of 21 ms each.
Let's again assume that 4 cameras are to be set up in synchronous free run mode. Camera 1 finishes the first exposure after 4 ms. Then follows the first readout time, which is 21 ms.
The second exposure starts after 25 ms and finishes after 45 ms.
Since now camera 1 has disabled its LEDs, camera 2 can already start exposure after 46 ms. (1 ms has been added as safety margin.)

Thus, the start time for camera n can be calculated as follows:

```
Tn = (n - 1) x (Texp1 + Texp2 + Treadout)
```

The maximum synchronous trigger rate can be calculated as follows:

```
Fmax = 1 / (n x (Texp1 + Texp2 + Treadout))
```

Having 4 cameras, the maximum synchronous trigger rate would be:

```
Fmax = 1 / (4 x (4 ms + 20 ms + 21 ms) = 1 / 180 ms = 5.6 Hz
```

Therefore, the synchronous trigger rate can safely be set to 5 Hz for 4 cameras in HDR mode.

3.7.5 Configuring a Camera for Synchronous Free Run

First, all cameras that are supposed to run synchronously need to be configured. One way to achieve this is to use the following static method to find all cameras:

`CToFCamera::EnumerateCameras()`

Since this method returns a list of CameraDescription objects, CTOFcamera objects can be created from the descriptions. Then, each camera has to be configured for synchronous free run. This involves the following steps:

- enabling the trigger mode
- setting the trigger source to "SyncTimer"

Next, set up the cameras using the `setupCameras()` method from the synchronous free run sample.
(At this stage, not trigger delays will be configured yet.)

Example:

```
void Sample::setupCameras()
{

    // const size_t nBuffers = 3;  // Number of buffers to be used for grabbing.

    cout << "Searching for cameras ... " << endl << endl;

    m_CameraList = CToFCamera::EnumerateCameras();

    cout << "found " << m_CameraList.size() << " ToF cameras " << endl << endl;

    // Store number of cameras.
    m_NumCams = (int)m_CameraList.size();
    size_t camIdx = 0;

    // Initialize array with master/slave info.
    for (size_t i = 0; i < MAX_CAMS; i++)
    {
        m_IsMaster[i] = false;
    }

    CameraList::const_iterator iterator;

    // Iterate over list of cameras.
    for (iterator = m_CameraList.begin(); iterator != m_CameraList.end(); ++iterator) {
        CameraInfo cInfo = *iterator;
        cout << "Configuring Camera " << camIdx << " : "<< cInfo.strDisplayName << "." << endl;

        // Create shared pointer to ToF camera.
        shared_ptr<CToFCamera> cam(new CToFCamera());

        // Store shared pointer for later use.
        m_Cameras.push_back(cam);

        // Open camera with camera info.
        cam->Open(cInfo);

        //
        // Configure camera for synchronous free run.
        // Do not yet configure trigger delays.
        //

        // Enable IEEE1588.
        CBooleanPtr ptrIEEE1588Enable = cam->GetParameter("GevIEEE1588");
        ptrIEEE1588Enable->SetValue(true);

        // Enable trigger.
        GenApi::CEnumerationPtr ptrTriggerMode = cam->GetParameter("TriggerMode");

        // Set trigger mode to "on".
        ptrTriggerMode->FromString("On");

        // Configure the sync timer as trigger source.
        GenApi::CEnumerationPtr ptrTriggerSource = cam->GetParameter("TriggerSource");
        ptrTriggerSource->FromString("SyncTimer");

        // Proceed to next camera.
        camIdx++;
    }

}
```

3.7.6 Finding the Master Device

Once IEEE1588 has been enabled for all ToF cameras, the cameras will start to negotiate which one should be the master clock. This happens transparently in the PTP layer without any further user interaction.
While negotiating, the user can check the status of the camera by reading the GevIEEE1588StatusLatched parameter. Before doing so, the status needs to be latched by executing the GevIEEE1588DataSetLatch parameter.

Possible return values for the status:

- Listening
- Master
- Slave

As long as the status returned by the node is "Listening", the negotiation between the cameras has not been completed.
The following excerpt from the SyncFreerun code sample shows how to retrieve the status from the camera.
This procedure is repeated for as long as the status is still "Listening".

```
// Latch IEEE1588 status.
CCommandPtr ptrGevIEEE1588DataSetLatch = m_Cameras.at(i)->GetParameter("GevIEEE1588DataSetLatch");
ptrGevIEEE1588DataSetLatch->Execute();

// Read back latched status.
GenApi::CEnumerationPtr ptrGevIEEE1588StatusLatched = m_Cameras.at(i)->GetParameter("
      GevIEEE1588StatusLatched");
// The smart pointer holds the node, not the value.
// The node value is always up to date.
// Therefore, there is no need to repeatedly retrieve the pointer here.
while (ptrGevIEEE1588StatusLatched->ToString() == "Listening")
{
    // Latch GevIEEE1588 status.
    ptrGevIEEE1588DataSetLatch->Execute();
    // Print dots to denote ongoing negotiation between cameras
    cout << "." << std::flush;
    mSleep(1000);
}
```

The value of the GevIEEE1588Status parameter is checked for each camera. The algorithm waits until the value changes from "Listening" to either "Master" or "Slave".

After it has been decided which camera is master (if any) and which cameras are slaves, the roles are stored in the m_master[] array for each camera. If the camera has acquired the master role, a '1' is stored in the array, otherwise '0'.

Note that if a PTP master clock is present in the subnet, the TOF cameras in the subnet all become slaves.

3.7.7 Synchronizing Cameras

Synchronization of the PTP clocks in the cameras is performed automatically within the PTP layer. Before starting image acquisition, it is mandatory to make sure that all slave clocks are in sync with the master clock.

Progress of synchronisation with the master clock can be checked by reading the GevIEEE1588OffsetFromMaster parameter.

```
CIntegerPtr ptrGevIEEE1588OffsetFromMaster = m_Cameras.at(camIdx)->GetParameter("
      GevIEEE1588OffsetFromMaster");
```

Before reading the node, the IEEE1588 status needs to be latched:

```
ptrGevIEEE1588DataSetLatch->Execute();
```

In the SynchFreerun code sample, the `syncCameras()` method is used to check by how much the slave clocks of each camera in slave role deviate from the master clock. The method will wait until all deviations are lower than the threshold specified.

Repeated readout of the GevIEEE1588OffsetFromMaster parameter value is implemented in the following helper method:

```
GetMaxAbsGevIEEE1588OffsetFromMasterInTimeWindow(...)
```

This is called from the `syncCameras()` method. This method returns the absolute deviation from master clock in nanoseconds.

3.7.8 Setting Trigger Delays

As explained above, a delay has to be configured between the image acquisitions of the individual cameras to avoid mutual disturbance.

A delayed acquisition, started at a certain point of time in the future (seen from the current time of the internal PTP clock of the camera) can be achieved as follows:
The current time is read from the PTP clock. It is returned in the "TimestampLow" and "TimestampHigh" parameters, which together form a 64-bit value representing the time elapsed in microseconds since the start of the clock.

The required delay is added to the timestamp and the result is split up into two words of 32 bits. These are written to the "SyncStartLow" and "SyncStartHigh" parameters.

The calculation of the timestamps is carried out following the method explained in the previous section.

Since meanwhile the PTP clocks of the cameras are synchronized, the timestamp of an arbitrary camera (always the first one that was enumerated in the beginning) is read out and used as the base for the calculation of delayed timestamps.

In the code sample, the proper trigger delay for each camera is calculated and written to the camera using the method `setTriggerDelays()` method.

Example:

```
void Sample::setTriggerDelays() {

    // Current timestamp
    uint64_t timestamp, syncStartTimestamp;

    // The low and high part of the timestamp
    uint64_t tsLow, tsHigh;

    // Initialize trigger delay.
    m_TriggerDelay = 0;

    cout << endl << "configuring start time and trigger delays ..." << endl << endl;

    //
    // Cycle through cameras and set trigger delay.
    //
    for (size_t camIdx = 0; camIdx < m_Cameras.size(); camIdx++)
    {

        cout << "Camera " << camIdx << " : " << endl;
```

```
    //
    // Read timestamp and exposure time.
    // Calculation of synchronous free run timestamps will all be based
    // on timestamp and exposure time(s) of first camera.
    //
    if (camIdx == 0)
    {
        // Latch timestamp registers.
        CCommandPtr ptrTimestampLatch = m_Cameras.at(camIdx)->GetParameter("TimestampLatch");
        ptrTimestampLatch->Execute();

        // Read the two 32 bit halves of the 64 bit timestamp.
        CIntegerPtr ptrTimeStampLow(m_Cameras.at(camIdx)->GetParameter("TimestampLow"));
        CIntegerPtr ptrTimeStampHigh(m_Cameras.at(camIdx)->GetParameter("TimestampHigh"));
        tsLow = ptrTimeStampLow->GetValue();
        tsHigh = ptrTimeStampHigh->GetValue();

        // Assemble 64-bit timestamp and keep it.
        timestamp = tsLow + (tsHigh << 32);
        cout << "Reading time stamp from first camera. \ntimestamp = " << timestamp << endl << endl;

        cout << "Reading exposure times from first camera:" << endl;

        // Get exposure time count (in case of HDR there will be 2, otherwise 1).
        CIntegerPtr ptrExposureTimeSelector = m_Cameras.at(camIdx)->GetParameter("ExposureTimeSelector"
);
        size_t n_expTimes = 1 + (size_t) ptrExposureTimeSelector->GetMax();

        // Sum up exposure times...
        CFloatPtr ptrExposureTime = m_Cameras.at(camIdx)->GetParameter("ExposureTime");
        for (size_t l = 0; l < n_expTimes; l++)
        {
            ptrExposureTimeSelector->SetValue(l);
            cout << "exposure time " << l << " = " << ptrExposureTime->GetValue() << endl << endl;
            m_TriggerDelay += (int64_t) (1000 * ptrExposureTime->GetValue());   // Convert from us ->
 ns
        }

        cout << "Calculating trigger delay." << endl;

        // Add readout time.
        m_TriggerDelay += (n_expTimes -1) * m_ReadoutTime;

        // Add safety margin for clock jitter.
        m_TriggerDelay += 1000000;

        // Calculate synchronous trigger rate.
        cout << "Calculating maximum synchronous trigger rate ... " << endl;
        m_SyncTriggerRate = 1000000000/(m_NumCams * m_TriggerDelay);

        // If the calculated value is greater than the maximum supported rate,
        // adjust it.
        CFloatPtr ptrSyncRate(m_Cameras.at(camIdx)->GetParameter("SyncRate"));
        if ( m_SyncTriggerRate > ptrSyncRate->GetMax() )
        {
            m_SyncTriggerRate = (uint64_t) ptrSyncRate->GetMax();
        }

        // Print trigger delay and synchronous trigger rate.
        cout << "Trigger delay = " << m_TriggerDelay/1000000 << " ms" << endl;
        cout << "Setting synchronous trigger rate to " << m_SyncTriggerRate << " fps"<< endl << endl;
    }

    // Set synchronization rate.
    CFloatPtr ptrSyncRate(m_Cameras.at(camIdx)->GetParameter("SyncRate"));
    ptrSyncRate->SetValue((double) m_SyncTriggerRate);

    // Calculate new timestamp by adding trigger delay.
    // First camera starts after triggerBaseDelay, nth camera is triggered
    // after a delay of triggerBaseDelay +  n * triggerDelay.
    syncStartTimestamp = timestamp + m_TriggerBaseDelay + camIdx * m_TriggerDelay;

    // Disassemble 64 bit timestamp.
    tsHigh = syncStartTimestamp >> 32;
    tsLow = syncStartTimestamp - (tsHigh << 32);

    // Get pointers to the two 32-bit registers, which together hold the 64-bit
    // synchronous free run start time.
    CIntegerPtr ptrSyncStartLow(m_Cameras.at(camIdx)->GetParameter("SyncStartLow"));
    CIntegerPtr ptrSyncStartHigh(m_Cameras.at(camIdx)->GetParameter("SyncStartHigh"));

    ptrSyncStartLow->SetValue(tsLow);
    ptrSyncStartHigh->SetValue(tsHigh);

    // Latch synchronization start time & synchronization rate registers.
    // Until latching the values, they won't be effective!
```

```
        CCommandPtr ptrSyncUpdate = m_Cameras.at(camIdx)->GetParameter("SyncUpdate");
        ptrSyncUpdate->Execute();

        // Show new synchronous free run start time.
        cout << "Setting Sync Start time stamp" << endl;
        cout << "SyncStartLow = " << ptrSyncStartLow->GetValue() << endl;
        cout << "SyncStartHigh = " << ptrSyncStartHigh->GetValue() << endl << endl;
    }
}
```

3.7.9 Synchronized Acquisition

After trigger delays have been written to the cameras, acquisition can be prepared. In the code sample, this is done in the `grabImages()` method.

Since a custom grab loop is employed again in this sample, the concepts described in the custom grab loop chapter apply here, too.

Once the grab loop has been set up, acquisition is started. In a round-robin scheme, one image is grabbed from each camera and the `processData()` method is called.
Further processing of the captured image data is handled here.

The following excerpt of the code sample shows the part where acquisition is prepared and then started. Note how several methods of the `CToFCamera` utility class are used:

```
    SetBufferAllocator()                // sets the buffer allocator
    PrepareAcquisition()                // allocates memory buffers and prepares iamge acquisitior
    QueueBuffer(i)                      // enqueues one buffer
    StartAcquisition()                  // starts the acqusition engine
    IssueAcquisitionStartCommand()      // starts image acquisition
```

```
 // Prepare cameras and buffers for image exposure.
for (size_t camIdx = 0; camIdx < m_NumCams; camIdx++)
{
    // Let the camera class use our allocator.
    // When the application doesn't provide an allocator, a default one that allocates memory buffers
    // on the heap will be used automatically.
    m_Cameras.at(camIdx)->SetBufferAllocator( new CustomAllocator(), true); // m_Camera takes ownership and
       will clean-up allocator.

    // Allocate the memory buffers and prepare image exposure.
    m_Cameras.at(camIdx)->PrepareAcquisition( nBuffers );

    // Enqueue all buffers to be filled with image data.
    for ( size_t j = 0; j < nBuffers; ++j )
    {
       m_Cameras.at(camIdx)->QueueBuffer( j );
    }

    // Start the acquisition engine.
    m_Cameras.at(camIdx)->StartAcquisition();

    // Now, the acquisition can be started on the camera.
    m_Cameras.at(camIdx)->IssueAcquisitionStartCommand(); // The camera continuously sends data now.
}
```

Processing Image Data

If no timeout or general error has occurred, the `processData( camIdx, grabResult )` method is called.

Once image data has been extracted from the image buffer in the `processData()` method, the buffer is returned to the image acquisition process by calling the `QueueBuffer()` method of the `CToFCamera` class.

Code Excerpt:

```
// A successfully grabbed buffer can be processed now. The buffer will not be overwritten with new data
      until
// it is explicitly placed in the acquisition engine's input queue again.
if ( grabResult.status == GrabResult::Ok )
{
    processData( camIdx, grabResult );
}

// Data processing has finished. Put the buffer back into the acquisition
// engine's input queue to be filled with new image data.
if ( grabResult.status != GrabResult::Timeout)
{
    m_Cameras.at(camIdx)->QueueBuffer( grabResult.hBuffer );
}
```

Note that within the `processData()` method, the status flag of the grabResult is checked before trying to access image data. (Otherwise the grab result and its data members may not be valid.)

In the code sample, the timestamp is retrieved from the image data and printed. By comparing the timestamps, the user can see how the images are acquired with the delay between the cameras.

Code Excerpt

```
void Sample::processData(size_t camIdx,  const GrabResult& grabResult )
{
    // Do not try to access data members of the grab result when a timeout has occurred
    // because the grab result doesn't represent a valid buffer in that case.
    // Also, in case of a failed grab, the buffer and the grab result members may
    // not be valid.
    if ( grabResult.status != GrabResult::Timeout )
    {
        uint64_t timeStamp = grabResult.timeStamp;
        uint64_t frameID = grabResult.frameID;
        if ( 0 == (frameID % 10) )
        {
            // Display timestamp in milliseconds.
            long double timeStampMillis = timeStamp / 1000000.0L;
            cout << fixed << setprecision(6) << "camera " << camIdx << "\tframeID = "
                << frameID << "\ttimestamp = " << timeStampMillis << " ms" << endl;
        }
    }

}
```