GigEConnect SDK

Sample Grabbing Code Walkthrough

Prerequisites

- In order to compile and link your application, you'll need the following items:
  - GenTL include files (Include Client.h)
  - TLI.lib import library to link against
  - TLI.dll grabber DLL
  - GenAPI includes and header files

Function return value

- Every library function returns a status of type GC_ERROR, which indicates whether it completed successfully. In the case of success, the code returned is GC_ERR_SUCCESS (0)
- Any other return value indicates some sort of failure during the function processing.
- In this walkthrough, we use the local variable gc defined as follows:

GC_ERROR gc;

Namespaces

- Following the GenICam model, the library defines several namespaces where all functions, variables and data types are defined.
- For the sake of this walkthrough, we do not use namespace prefixes, but instead use the following code immediately following the #include directive(s):

```
using namespace GenICam;
using namespace GenICam::Client;
using namespace GenApi;
```

Opening the library

- In order to open the library, your code must call TLOpen().
- The sole parameter is a pointer to variable that will hold a handle to the library.
- The code:

```
TL_HANDLE hTL;
gc = TLOpen( &hTL );
```

Enumerating interfaces

- The first step is to get the number of interfaces available. This is achieved by calling TLGetNumInterfaces():

```
uint32_t num_ifaces;
gc = TLGetNumInterfaces( hTL, &num_ifaces  );
```

Enumerating interfaces

- Next step is getting an interface name through the call to TLGetInterfaceName():

```
char iface_name[256];
size_t cb = sizeof(iface_name);
gc = TLGetInterfaceName(hTL, 0, iface_name, &cb );
```

Please note that we want to get the name of interface 0 (the first one).

Opening interface

- To open an interface, we call TLOpenInterface() specifying the handle to the library and the interface name

```
INTERFACE_HANDLE hIF;
gc = TLOpenInterface( hTL, iface_name, &hIF );
```

Enumerating devices

- Enumerating devices on an interface is pretty much like enumerating interfaces. First we call TLGetNumDevices():

```
uint32_t num_devices;
gc = TLGetNumDevices( hIF, &num_devices );
```

Enumerating devices

- Next, we get the name of a device with specified index (0 in our case) by calling TLGetDeviceName():

```
char dev_name[256];
gc = TLGetDeviceName( hIF, 0, dev_name,
        sizeof(dev_name) );
```

Opening device

- After getting both the interface and the device names, we are ready to open the device. This is done through a call to TLOpenDevice() as illustrated below:

```
DEV_HANDLE hDev;
gc = TLOpenDevice( hIF, device_name, &hDev );
```

Accessing device registers

- The device is controlled by the means of writing its registers.
- Getting device parameters involves reading its registers.
- In order to achieve device independence as well as make the access much less cumbersome, the access is performed through string register names as opposed to numeric register addresses.

Accessing device registers

- The library supplies two functions to read/write device registers: GCReadData() and GCWriteData().
- These functions are a part of the low-level interface and consequently expect specifying register addresses.
- However we don’t use these functions directly, but instead work with GenAPI library, which will effectively translate string names to the addresses corresponding to these registers.

Accessing device registers

- To access device registers, GenAPI library requires two things:
  - User-created port object, whose methods will be called by GenAPI to perform actual data transfer by calling GCReadData() and GCWriteData()
  - Register description XML file

Accessing device registers

- First, we create a port object that will be called by GenAPI to perform actual register I/O. We declare a class that implements IPort interface and implements its methods Read() and Write().
- The class is declared as follows:

Acessing device registers

```
class MyPort: public IPort
{
public:

        MyPort( DEV_HANDLE hDev ):
          m_hDev(hDev)
          {
          };

protected:

        virtual void Read(void *pBuffer, int64_t Address, int64_t Length)
        {
                if( GC_ERR_SUCCESS != GCReadData( m_hDev, Address, pBuffer, (size_t*)&Length ) )
                        throw ACCESS_EXCEPTION( "Failure reading device data" );
        };

   virtual void Write(const void *pBuffer, int64_t Address, int64_t Length)
        {
                if( GC_ERR_SUCCESS != GCWriteData( m_hDev, Address, pBuffer, (size_t*)&Length ) )
                        throw ACCESS_EXCEPTION( "Failure writing device data" );
        };

        virtual EAccessMode GetAccessMode() const
        {
                return RW;
        };

private:
        DEV_HANDLE m_hDev;
};
```

Acessing device registers

- The Read() and Write() methods do the actual work of reading/writing device registers. We just call the corresponding library functions to get it done.
- The GetAccessMode() method is required by GenAPI to get the port access mode. We return RW (Read/Write)

Acessing device registers

- After declaring our port class, the next step is to get an XML describing device registers, so GenAPI can correctly translate register name to addresses. This is achieved by first calling DevGetURL() to obtain the localtion of the file:

```
char url[2048];
size_t url_len = sizeof(url);
gc = GCGetURL( hDev, url, &url_len );
```

Accessing device registers

- Now, that we have the register description file URL, we should read the file contents
- In our example, we assume that the returned URL always has the file: prefix. Other variants include http:, ftp: and local:. So, we extract the file path:

```
char* file_name = url + (sizeof("file:")-1);
```

Accessing device registers

- Then, we open the file and read all its contents as binary data:

```
FILE* pf = fopen( file_name, "rb" );
size_t file_len = filelength( pf->_file );
char* xml = new char[file_len+1];
fread( xml, 1, file_len, pf );
fclose( pf );
```

- Since the XML file contains string data, we terminate the data with a NULL char:

```
xml[file_len] = 0;
```

Acessing device registers

- Next, we create an instance of the GenAPI NodeMap object and load the XML

```
CNodeMapRef node_map;
node_map._LoadXMLFromString( xml );
```

- Then, we create an instance of our port object and connect the map to it

```
MyPort port( hDev );
node_map._Connect( &port, "CameraPort" );
```

- Now, GenAPI is ready to be used for accessing our device's registers

Creating data stream

- Having the device open and register access ready, we are now creating a data stream required for image acquisition. This is done through DevCreateDataStream():

```
DS_HANDLE hDS;
gc = DevCreateDataStream( hDev, 0, &hDS  );
```

- Please note that only one data stream is currently supported.

Allocating image buffers

- Image acquisition required allocating several buffers that will be filled with incoming image data. Each buffer should also be announced and queued to be ready for use.
- An important point is calculating buffer size. The size depends on 3 parameters: image width, image height and pixel format. All these are device parameters that should be read using GenAPI NodeMap object described before:

```
CIntegerPtr pWidth = node_map._GetNode("Width");
CIntegerPtr pHeight = node_map._GetNode("Height");
CEnumerationPtr pPixelFormat = node_map._GetNode("PixelFormat");
```

Allocating image buffers

- After having discovered these 3 parameters, the required buffer size is calculated as Width * Height * Factor, where the factor depends on the pixel format: for RGB8 the factor is 3, for Mono8, the factor is 1, for Mono10 and Mono12, the factor is 1.5 and for Mono16, the factor is 2.

Allocating image buffers

- Having calculated the buffer size, we should allocate a buffer and announce it to the grabber using DSAnnounceBuffer():

```
PVOID pBuffer;
BUFFER_HANDLE hBuffer;
pBuffer = new BYTE[buf_size];
gc = DSAnnounceBuffer( hDS, pBuffer, buf_size, NULL,
                       &hBuffer );
```

Allocating image buffers

- Next, the announced buffer must be queued by calling DSQueueBuffer():

gc = DSQueueBuffer(hDS, hBuffer);

- The number of buffers is specified by the application and is usually user-configurable.

Registering image arrival event

- Before image acquisition starts, we must register an event, which will be fired each time an image is acquired. This is done through GCRegisterEvent():

```
HANDLE hEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
EVENT_HANDLE hEvt;
gc = GCRegisterEvent( hDS, EVENT_NEW_BUFFER, hEvent, &hEvt );
```

- The Windows event object is used for actual wait by WaitForSingleObject, as we'll see later

Starting acquisition

- Starting acquisition involves two steps:
  - Telling the grabber to start acquisition
  - Telling the device to start video transmission
- The first is done by calling DSStartAcquisition():

```
gc = DSStartAcquisition( hDS );
```

- The second is achieved by issuing a command to the device:

```
CCommandPtr pStart = node_map._GetNode("AcquisitionStart");
pStart->Execute();
```

Grabbing images

- To grab incoming video images, we wait on the Windows event object specified when we have registered the event:

  WaitForSingleObject(hEvent, 10000);

- Once the event is signalled, we retrieve the handle of the buffer that contains the acquired image:

```
BUFFER_HANDLE hBuf;
cb = sizeof(hBuf);
gc = GCGetEventData( hEvt, &hBuf, &cb );
```

Grabbing images

- After we have used the image, e.g. have it drawn on the screen, we must requeue the buffer for use by the grabber again:

gc = DSQueueBuffer(hDS, hBuf);

Stopping acquisition

- Similar to starting acquisition, stopping it involves two steps:
  - Telling the device to stop transmitting video
  - Telling the grabber to stop acquisition
- The first is achieved by executing a device command:

```
CCommandPtr pStop = node_map._GetNode("AcquisitionStop");
pStop->Execute();
```

- The second is done by calling DSStopAcquisition():

```
gc = DSStopAcquisition( hDS );
```

Cleaning up

- After we are done with the device, we perform the following steps:
  - Close data stream:

    gc = DSClose(hDS);

  - Close device:

    gc = DevClose(hDev);

  - Close the library:

    gc = TLClose(hTL);