GenICam

CLProtocol Module

Using GenApi with CameraLink

Table of Contents

HISTORY

Version	Date	Changed by	Change
1.0	08.12.2009	Fritz Dierks, Basler	First Draft

1 Overview

This module of the GenICam standard describes how to configure CameraLink[*] using the GenApi module of the GenICam standard. The problem with CameraLink is that it does not define cameras to be register based. Instead the CameraLink configuration interface is based on an **ISerial** interface with allows to send and receive blocks of bytes. The GenApi module however requires an **IPort** interface which allows to get and set registers in the camera.

The CLProtocol module defines the interface of a CameraLink protocol driver DLL which must be provided by the camera manufacturer. The DLL must implement an IPort interface using the ISerial interface as connection to the camera (see Figure 1).

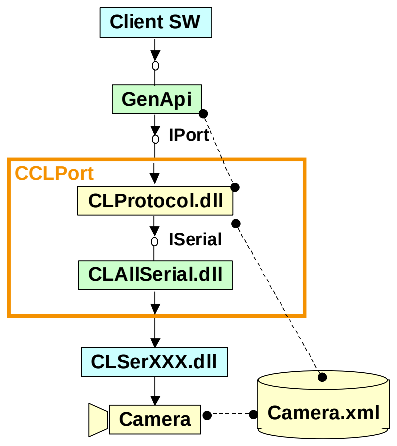
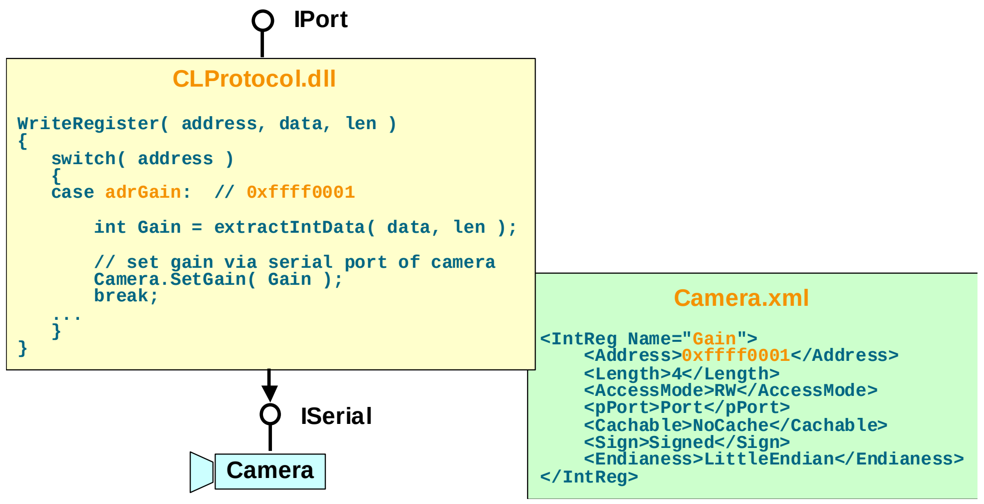


Figure 1 Using the CLProtocol.dll to configure a camera

If a camera is natively register based the CLProtocol DLL is just a simple protocol driver running for example a binary register access protocol like the CANbus protocol. If however a camera is for example ASCII based the CLProtocol driver DLL must implement a pseudo register space and provide the corresponding camera description XML file (see Figure 3).



```
WriteRegister( address, data, len )
{
   switch( address )
   {
   case adrGain:  // 0xffff0001

       int Gain = extractIntData( data, len );

       // set gain via serial port of camera
       Camera.SetGain( Gain );
       break;
   ...
   }
}
```

```
<IntReg Name="Gain">
    <Address>0xffff0001</Address>
    <Length>4</Length>
    <AccessMode>RW</AccessMode>
    <pPort>Port</pPort>
    <Cachable>NoCache</Cachable>
    <Sign>Signed</Sign>
    <Endianess>LittleEndian</Endianess>
</IntReg>
```

Figure 2 Providing a pseudo register space

This **CLProtocol driver DLL** has a pure C interface and is normally not used directly. Instead the **GenICam reference implementation** provides a C++ wrapper class **CCLPort** deals with tasks like loading and binding of the best matching driver DLL (see Figure 1). The wrapper class uses the **CLAllSerial.dll / CLSerXXX.dll**

[*] CameraLink standard v1.1 provided by the Automated Imaging Association (AiA)

mechanism[†] defined by the CameraLink standard to communicate with the camera. Note that the wrapper class is not part of the standard. For more details refer to the CLProtocol tutorial coming with the reference implementation.

The C-interface of the CLProtocol driver DLL is not operating dependent, however the compiled DLL is. Currently the following operating systems are supported:

- Windows XP (Win32) or higher : this operating system MUST be supported
- Windows XP (Win64) or higher : this operating system SHOULD be supported

In order to setup and use a CLProtocol DLL the following steps must be performed:

1. The CLProtocol driver DLLs and any accompanying XML files must be installed and registered in the system
2. For each frame grabber port the right CLProtocol DLL must be selected and the camera must be identified
3. A camera description XML file must be retrieved

These steps are described in the following sections. In addition the ISerial interface and the C functions forming the interface of the CLProtocol driver DLL are explained. Last but not least there is a section on how to handle the baud rate.

2 Installing and Registering CLProtocol DLLs

The CLProtocol driver DLLs and any corresponding XML files can be installed by the camera vendor's setup program to an arbitrary location on the target machine, e.g.:

```
c:\program files\MyVendor\CLProtocol
```

XML files accompanying the DLLs are installed directly to that location. For each supported operating system there is a separate sub-directory with a name defined by this standard were the corresponding DLL must be installed to. For Win32 the sub-directory's name is **Win32_i86**; for Win64 the name is **Win64_x86**. Here is an example:

```
c:\program files\MyVendorDir\CLProtocol              # install XML files here
c:\program files\MyVendorDir\CLProtocol\Win32_i86    # install Win32 DLL here
c:\program files\MyVendorDir\CLProtocol\Win64_x64    # install Win64 DLL here
```

Multiple DLLs with different names can reside in one sub-directory. The DLL name must be of the form ***.dll**.

The registration is performed by adding the location (i.e. the directory name without trailing backslash) to a list of locations given in the environment variable **GENICAM_CLPROTOCOL**, for example:

```
GENICAM_CLPROTOCOL=c:\program files\MyVendorDir\CLProtocol;c:\temp\MyTest
```

If the environment variable does not exist it must be created.

If the DLLs are uninstalled the location entry must be removed from the list of locations leaving any others in place. If no other entry is left the environment variable must be deleted.

While a CLProtocol driver DLL is under development it can be compiled in Debug or Release mode. In order to simplify the life of the developer the Debug version of a DLL named XXX.dll should be named XXX.debug.dll. The following rules apply when the DLLs are enumerated by the CCLPort helper class:

- If the CCLPort helper class is compiled in **Debug** mode a DLL named XXX.dll is loaded only if there is no corresponding DLL named XXX.debug.dll in the same directory.
- If the CCLPort helper class is compiled in **Release** mode a DLL named XXX.debug.dll is loaded only if there is no corresponding DLL named XXX.dll in the same directory.

[†] Note that for Win64 the naming scheme is CLAllSerial_w64.dll / CLSerXXX_w64.dll where XXX is a 3 letter abbreviation of the frame grabber vendor's name.

3 Selecting a CLProtocol DLL and Identifying a Camera

The key problem when setting up the DLL is to identify the manufacturer and model name of the camera connected to a frame grabber port. This information is required in order to select the right CLProtocol DLL but it is also required by the driver DLL itself for adapting its behavior to different camera models of the same vendor.

It would be nice if the manufacturer as well as the model name of an arbitrary CameraLink camera could be determined automatically just by probing the frame grabber port. However this kind of plug&play mechanism will stay a dream for CameraLink because for historical reasons there is no standard protocol for the serial port of CameraLink cameras and cameras of different vendors can behave very differently. Probing a camera with different protocol variants would take too long and could even drive some camera models in an undefined state from which they would recover.

So there is no way around the user selecting at least the camera manufacturer name and thus the CLProtocol DLL for each frame grabber port manually. Having done that the DLL can identify the camera in a more or less automatic way because the vendors should know their cameras well enough in order to automate that task. Nevertheless it that automation is not possible for some reason the standard provides means to deal with that situation, too.

The whole identification process is based on string identifiers (IDs) which are enumerated by the system and (partially) selected by the customers.

DeviceID

The identifier resulting from the camera identification process is called the **DeviceID**. It contains all data required to uniquely identify a device and its corresponding CLProtocol driver DLL. This data is assembled in a string which is composed of tokens separated by the hash ('#') sign:

```
"DriverDirectory#DriverFileName#Manufacturer#Family#Model#Version#SerialNumber"
```

The first two tokens describe the **directory** where the protocol driver DLL is found (without trailing back slash) and the **file name** of the DLL. The other tokens are from left to right the camera's **manufacturer**, **family**, **model, version**, and **serial number**. Each of these latter tokens must follow the naming convention for C variables, i.e. they must match the following regular expression:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Either the serial number or the serial number and the version token can be omitted. Here two examples for valid DeviceIDs:

```
"c:\program files\MyVendorDir\Win32_i86#MyDriver.dll#MyVendor#MyFamily1#MyModelA#Ver_2a#SerNo123"
"c:\program files\MyVendorDir\Win32_i86#MyDriver.dll#MyVendor#MyFamily1#MyModelA"
```

DeviceID Templates

In order to address a subset of possible DeviceIDs a **DeviceID template** can be formed by the DeviceID from the right up to but not including the manufacture name.

For example in order to address all cameras of a certain family the corresponding DeviceID template would looks like this:

```
"c:\program files\MyVendorDir#MyDriver.dll#MyVendor#MyFamily1"
```

A DeviceID template is said to **match** an DeviceID if the left part of the DeviceID string is identical to the DeviceID template.

For example the template given above would match the following DeviceIDs

```
"c:\program files\MyVendorDir#MyDriver.dll#MyVendor#MyFamily1#MyModelA#Version_2a#SerNo234"
"c:\program files\MyVendorDir#MyDriver.dll#MyVendor#MyFamily1#MyModelB#Version_2b#SerNo432"
```

but not this one

```
"c:\program files\MyVendorDir#MyDriver.dll#MyVendor#MyFamily2#MyModelC#Version_2a#SerNo345"
```

because the family is different.

Short DeviceID (Templates)

A short DeviceID or short DeviceID template is just a original string with the first two items – the DLL directory and file name including the trailing hash sign – missing. For example if a DeviceID template reads

```
"c:\program files\MyVendorDir\Win32_i86#MyDriver.dll#MyVendor#MyFamily1"
```

the corresponding short DeviceID is

```
"MyVendor#MyFamily1"
```

Probing a Device

Ideally a customer being about to setup a frame grabber port is just presented a list of all CLProtocol DLLs installed in the system, each being represented by the corresponding manufacturer name. However it may not be possible for each DLL to fully automatically identify the camera attached to the selected port. For those cases the CLProtocol DLL provides a list of DeviceID templates for the user to select one.

For example the CLProtocol DLL of a VendorA might be able to deal with two camera families Family1 and Family2 but for example might no be able to automatically distinguish between cameras of the two families, because they implement very different protocols. In this case VendorA’s CLProtocol driver DLL would supply the following two DeviceID templates:

```
"c:\program files\MyVendorDir#MyDriver.dll#VendorA#Family1"
"c:\program files\MyVendorDir#MyDriver.dll#VendorA#Family2"
```

A VendorB whose DLL can do a fully automated detection of all cameras would only supply a single DeviceID template like this:

```
"c:\program files\MyVendorDir#MyDriver.dll#VendorB"
```

A VendorC however might have not bothered with automatic identification altogether and just enumerates all camera models the DLL can deal with:

```
"c:\program files\MyVendorDir#MyDriver.dll#VendorC#Family1#ModelX"
"c:\program files\MyVendorDir#MyDriver.dll#VendorC#Family1#ModelY"
"c:\program files\MyVendorDir#MyDriver.dll#VendorC#Family2#ModelZ"
```

In a system were CLProtocol DLLs from vendors A, B, and C are installed at the same time the user setting up a frame grabber port would get presented the following list of short DeviceID templates to select one:

```
"VendorA#Family1"
"VendorA#Family2"
"VendorB"
"VendorC#Family1#ModelX"
"VendorC#Family1#ModelY"
"VendorC#Family2#ModelZ"
```

After the user has selected a DeviceID template the CLProtocol driver DLL should be able to **probe** and **identify** the attached camera using the DeviceID template as a **hint**. If the identification is successful the CLProtocol driver DLL returns a full DeviceID string unambiguously identifying the camera found connected to the port.

PortIDs

Before the probing can take place the user has to select a port. The ports are enumerated using the **CLAllSerial.dll** and the result is presented in form of a list of **PortID** strings unambiguously identifying a port in the system.

The CLAllSerial.dll first enumerates all CLSerXXX DLLs found installed in the system, then it enumerates all frame grabber boards per DLL and finally all port per frame grabber board (see Figure 3). The PortID system however hides this enumeration hierarchy and presents the result of the enumeration process as a flat list of PortIDs.

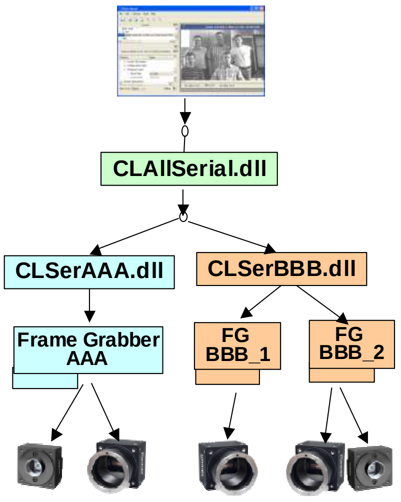


Figure 3 How the CLAllSerial.dll enumerates frame grabber ports

A **PortID** is a string of the following form:

```
"FrameGrabberManufacturer#PortName"
```

The token on the left of the hash ('#') sign is the frame grabber's manufacturer name and the token to the right the port name. Both strings are retrieved via the **clGetPortInfo** function defined in the CameraLink standard.

If a CompanyZ has for example two frame grabbers installed in a system with two serial ports each the following list of PortIDs would be result:

```
"CompanyZ#BoardAPort1"
"CompanyZ#BoardAPort2"
"CompanyZ#BoardBPort1"
"CompanyZ#BoardBPort2"
```

The standard COM ports of a PC are available via a pseudo frame grabber manufacturer called "**COM_Port**" enumerating PortIDs of the following form:

```
"COM_Port#COM1"
"COM_Port#COM2"
etc.
```

The COM_Port frame grabber DLL comes as part of the reference implementation.

Another pseudo frame grabber is available named "Local" which is used for ISerial implementations provided statically without using the enumeration mechanism of the CLAllSerial.DLL. This may for example be used in embedded systems. In this case a PortID could for example look like this:

```
"Local#TheOneAndOnlyPort"
```

The COM_Port frame grabber DLL comes as part of the reference implementation.

Summary

The following list summarizes the steps a client program has to take in order to select a CLProtocol driver DLL and identify a camera connected to a frame grabber port.

1. Retrieve a list of PortIDs
2. Present the list of PortIDs to the user to select a frame grabber port for configuration
3. Retrieve a list of DeviceID templates for the selected port
4. Present the list of DeviceID templates to the user to select the best matching template
5. Probe the camera using the selected DeviceID template as a hint. If the camera is recognized a DeviceID is returned unambiguously identifying the camera attached to the selected port
6. Connect to the camera using the DeviceID as identifier.

7. Store the DeviceID for later re-connection.

4 Retrieving an XML File for a Camera

Once the CLProtocol driver DLL is set up and the connection to the camera is established a XML camera description must be retrieved either from the camera or from the file system.

Because there could be more than one matching XML description, e.g. referring to different GenApi schema versions, the standard provides a two step approach for retrieving the XML code: First a sorted list of possible XML descriptions is created, with the best matching description coming first.

Users relying on the automatic just always take the first description to create the GenApi XML node map and configure the camera. If the user wants more control however he can select another XML description manually thus overriding the automatic.

XML IDs

Each XML description is identified by a **XML ID** which has the following form:

```
"SchemaVersion.1.0@<shortDeviceID>@DeviceVersion.1.2.3"
```

The XML ID is composed of three tokens delimited by an at ("@") sign.

The first token describes the version number of the GenApi schema the XML description uses. It has the form

```
"SchemaVersion.<VersionMajor>.<VersionMinor>"
```

where <VersionMajor> and <VersionMinor> are integers.

The second token is a short DeviceID template. It thus can have one of the following forms

```
"Manufacturer"
"Manufacturer#Family"
"Manufacturer#Family#Model"
"Manufacturer#Family#Model#Version"
"Manufacturer#Family#Model#Version#SerialNumber"
```

The third token describes the version number given in the XML description file for the device. It has the form

```
"XMLVersion.<VersionMajor>.<VersionMinor>.<VersionSubMinor>"
```

where <VersionMajor>, <VersionMinor>, and <VersionMinor> are integers. Note that the Version from the DeviceID string is an arbitrary CName and not necessarily identical to the version given in the XML. This makes for example sense if a XML file for an existing camera is created stepwise, each step covering more for the camera’s functionality while the camera itself is not changing.

Here is an example for a XML ID denoting a XML description which is valid for a whole family of cameras

```
"SchemaVersion.1.1@MyVendor#MyFamily1@XMLVersion.1.2.3.xml"
```

The list of XML IDs is assembled from the following sources:

- The CLProtocol DLL checks which XML descriptions the camera can provide itself. In order to support this the camera might implement a **Manifest** register as described in the GigE Vision standard.
- The CLProtocol DLL itself might contain suitable XML description, e.g. compiled in as Windows resource.
- The directory containing the CLProtocol DLL may contains additional XML files. The name of these files must be **<XML ID>.xml**, e.g. `"SchemaVersion.1.0@MyVendor#MyFamily@XMLVersion.1.2.3.xml"`

Note that the retrieval of the XML files stored on the file system is performed by the reference implementation so the CLProtocol driver DLL does not have to implement that part.

If a XML ID is retrieved two immediate checks are made:

- If the SchemaVersion cannot be handle by the GenApi version used the XML ID is discarded.
- If the DeviceID template contained in the XML ID does not match the current DeviceID the XML ID is discarded as well.

Example 1: A XML ID

```
"SchemaVersion.1.2@CameraManufacturer@XMLVersion.1.2.3.xml"
```

would be rejected by GenICam v2.0 because that version can handle only schema versions v1.0 and v1.1.

Example 2 : If the DeviceID is `"MyVendor#Familiy1"` a XML ID

```
"SchemaVersion.1.2@MyVendor#Familiy2@XMLVersion.1.2.3.xml"
```

would not match (wrong family) and be discarded.

Finally the list of not rejected XML IDs is sorted according to the following rules:

- A higher SchemaVersion number goes first.
- Within the same SchemaVersion a longer DeviceID template goes first
- Within the same SchemaVersion and DeviceID template a higher DeviceVersion number goes first

Example:

```
"SchemaVersion.1.1@MyVendor#Familiy2@XMLVersion.1.2.0.xml"
"SchemaVersion.1.1@MyVendor#Familiy2@XMLVersion.1.0.0.xml"
"SchemaVersion.1.1@MyVendor@XMLVersion.3.0.0.xml"
"SchemaVersion.1.0@MyVendor@XMLVersion.3.0.0.xml"
```

The user can select a XML ID (possibly the fist one which is the best matching) and use this to retrieve the XML description itself. Using this description GenApi can then give access to the camera features.

Summary

The following list summarizes the steps a client program has to take in order to retrieve an XML description for an already connected camera.

1. Retrieve a sorted list of XML IDs
2. Optionally present the list to the user to select one. The default selection is the first and – due to the sorting – best matching XML ID
3. Retrieve the XML description associated with the selected XML ID

5 Standardized Programming Interfaces

The CLProtocol driver DLL must implement a set of C functions. The necessary header files are part of the standard.

- **CLProtocol.h** – declares the C functions to be implemented by the CLProtocol driver DLL
- **CLSerialTypes.h** – declares some types and constants
- **ISerial.h** – declares an abstract C++ interface ISerial which is used by the CLProtocol driver DLL to access the serial port. A C alias of the virtual function table formed by the C++ interface is also given so an implementation of the CLProtocol driver DLL in pure C is possible.

These header files contain a detailed description of the functions and their parameters which can be extracted using DoxyGen[‡]. This section gives an overview and explains how the functions are used.

5.1 *ISerial Interface*

The CLProtocol driver DLL need to have access to the frame grabber's serial port. This is given by a pointer to an **ISerial** interface which contains the following methods:

- **clSerialRead** – use this method to retrieve an array of bytes from the camera with timeout. The functionality and parameters are the same as with the corresponding function of the CameraLink standard.

[‡] A tool to create HTML documentation from C/C++ code commented using a special tags. See www.doxygen.org

- **clSerialWrite** – use this method to send an array of bytes to the camera with timeout.
- **clGetSupportedBaudRates** – this method provides the set of baud rates supported by the frame grabber board in form of a bit field.
- **clSetBaudRate** – this method sets the baud rate of the frame grabber board

The functionality and parameters of the four methods listed above are the same as with the corresponding function of the CameraLink standard. Because boards supporting only CameraLink v1.0 must be supported no advanced functions like GetNumBytesAvail are supported.

5.2 *CLProtocol Interface*

The functions to be implemented by the CLProtocol driver DLL are explained along the use cases introduced in the previous sections. Note that the client of the interface described here is typically the wrapper class **CCLPort** contained in the reference implementation and not the end user's code. However since the wrapper class is not part of the standard it would be possible by a user to write their own client code from scratch.

Retrieving a List of DeviceID Templates

The function **clpGetShortDeviceIDTemplates** is used to collect a list of DeviceID templates. The environment variable GENICAM_CLPROTOCOL contains a list of locations were CLProtocol driver DLL are stored. The wrapper class loads each of those DLLs and calls clpGetShortDeviceIDTemplates retrieving for each DLL a list of short DeviceID templates the respective DLL will understand. The combined short DeviceID templates decorated with the location of the DLL where they originate from for the desired list of DeviceID templates.

Note that for calling clpGetShortDeviceIDTemplates no ISerial interface needs to be supplied.

Probing, Identifying and Re-Connecting a Camera

The function **clpProbeDevice** is called with an ISerial interface and a DeviceID template as input parameter. The function attempts to identify the camera attached to the respective frame grabber port using the DeviceID template as hint. If the function is successful it returns a DeviceID as well as a Cookie (see below).

If the DeviceID is already known the function **clpProbeDevice** can also be used to re-connect the camera. This is simply done by handing in the DeviceID instead of a DeviceID template. It is the responsibility of the CLProtocol driver DLL to distinguish between the two use cases. Re-connecting instead of probing again makes sense because re-connecting is normally much faster than probing. As a general run an application should probe and identify a camera only once and then store the DeviceID for re-connect.

By calling clpProbeDevice a connection to the camera is opened. This connection is identified by the **Cookie** which must be handed in for any subsequent calls to the CLProtocol driver DLL. The DLL may use the Cookie to persist any data while the connection is open.

Closing a Connection to the Camera

In order to close the connection to the camera call **clpDisconnect** handing in the Cookie. On this call the DLL must free all persistent data attached to the connection and the Cookie becomes invalid.

Retrieving the XML Description for a Camera

Calling **clpGetXMLIDs** returns a list of XML IDs exposing what kind of XML descriptions the camera and/or the DLL itself is able to provide. Note that this does not include XML descriptions stored on the file system beside the DLL. These XML IDs belonging to these XML files are added by the wrapper class CCLPort.

The XML IDs returned from calling clpGetXMLIDs are not sorted. This is also done by the wrapper class.

If the user has selected a XML ID originating from the call to clpGetXMLIDs he can retrieve the actual XML description by calling **clpGetXMLDescription** handing in the XML ID as a parameter. Again, the wrapper class handles XML IDs belonging to XML files stored on the file system.

The best matching XML ID is typically only determined once and then stored along with the DeviceID for re-connection. The wrapper class caches XML descriptions retrieved during the first connect and thus makes sure that unnecessary XML file downloads from the camera are avoided.

Accessing Camera Registers

Camera Registers are read and written to using the functions **clpReadRegister** and **clpWriteRegister**. Both function calls require an ISerial interface, a Cookie and a timeout which should by default be set to 500ms.

For commands taking a longer time to complete than the typical timeout the function clpWriteRegister can return a special value **CL_ERR_PENDING_WRITE**. In this case the client code must call clpContinueWriteRegister which either completes the call or returns CL_ERR_PENDING_WRITE again. The function clpContinueWriteRegister can be called with a cancel flag thus abandoning the command processing. After cancelling a call the camera must be in a state to accept further commands without problem. The wrapper class handles the whole pending business under the hood so the customer will normally not notice it.

Error Handling

Each call to one of the DLL's functions returns an error code which normally will be CL_ERR_NO_ERR (=0). The error codes can origin from different places each living in a separate number range. A negative number indicates an error, a positive number a success.

- Standard error codes from the CLSerXXX interface definition: ±10***
- Standard error codes from the CLProtocol interface definition: ±20***
- Custom error codes from the CLProtocol implementations: ±30***

All other numbers are reserved

The CLProtocol driver DLL implements the function **clpGetErrorText** which when given a custom error code must return an error description message in English language. A similar function is also implemented by the CLSerXXX DLLs. If the wrapper class receives a negative return code is first asks the CLProtocol driver DLL for a error description text. If that call does not return a valid error message, it calls CLAllSerial DLL which in turn asks the CLSerXXX DLL and finally it tries to look-up a message text in a list of standard error messages.

Interface version

In order to prepare for future extensions of the CLProtocol DLL the function **clpGetCLProtocolVersion** must be implemented returning the major and minor version number of the interface. Different major version numbers make two protocols incompatible. A higher minor version number makes the interface backwards compatible to one with a lower minor version.

The current version number is major.minor = 1.0.

6 Handling the Baud Rate

A special feature of the camera is the **BaudRate**. It is special because when changed it must be changed in the camera and the frame grabber at the same time; otherwise connection to the camera is lost. The frame grabber's baud rate can be changed by the CLProtocol DLL via the CLAllSerial DLL's interface which also provides means to query a list of possible baud rates supported by the grabber.

GenICam therefore defines the BaudRate as standard feature which must be implemented by the CLProtocol DLL. Besides of the standard baud rates 9600 etc. a special **AutoMax** baud rate can be optionally implemented which is the maximum baud rate the camera and the frame grabber can run with.

The CLProtocol DLL should implement baud rate **auto detection**, i.e. being able to identify the camera's baud rate during probing.