GEN<i>CAM

# CLProtocol Module

Using GenApi with Camera Link®

**Version 1.2**

# Table of Contents

1 OVERVIEW ..... 3

2 INSTALLING AND REGISTERING CLPROTOCOL DRIVER LIBRARIES ..... 4

3 SELECTING A CLPROTOCOL DRIVER LIBRARY AND IDENTIFYING A CAMERA ..... 5

4 RETRIEVING AN XML FILE FOR A CAMERA ..... 9

5 HANDLING THE BAUD RATE ..... 10

6 STANDARDIZED PROGRAMMING INTERFACES ..... 11

6.1 ISERIAL INTERFACE ..... 11
6.2 CLPROTOCOL INTERFACE ..... 11

**HISTORY**

| Version | Date | Changed by | Change |
|---|---|---|---|
| 1.0 | 08.12.2009 | Fritz Dierks, Basler | First Draft |
| 1.0.1 | 30.08.2010 | Fritz Dierks, Basler | Added bootstrap registers |
| 1.1 | 04.06.2011 | Fritz Dierks, Basler | Added v1.1 extensions |
| 1.1.1 | 31.10.2011 | Fritz Dierks, Basler | Clarified baud rate handling |
| 1.1.2 | 08.07.2015 | Fritz Dierks, Basler | Clarified that in CCLPort initialize the cookie value is a reserved value |
| 1.2 | 09.07.2018<br>18.09.2018 | Silvio Voitzsch, Baumer<br>Christoph Zierl, MVTec | - Added Linux support<br>- Added new parameter to stop probing<br>- Added new function to deal with device events<br>- Fixed some spelling mistakes and added some clarifications<br>- Added enumeration procedure for CLSerXXX modules |

# 1 Overview

This module of the GenICam standard describes how to configure a Camera Link® camera using the GenApi module of the GenICam standard. The Camera Link specification does not define cameras to be register-based. Instead, the Camera Link configuration interface is based on an **ISerial** interface which allows sending and receiving blocks of bytes. The GenICam GenApi module however requires an **IPort** interface which allows getting and setting registers in the camera.

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

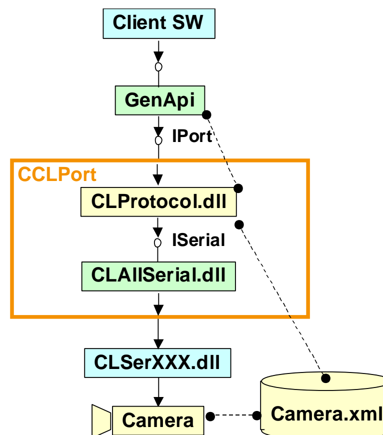


Figure 1 Using the CLProtocol driver library to configure a camera

If a camera is natively register-based the CLProtocol driver library 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 library must implement a pseudo register space and provide the corresponding camera description XML file (see Figure 2Figure 3).

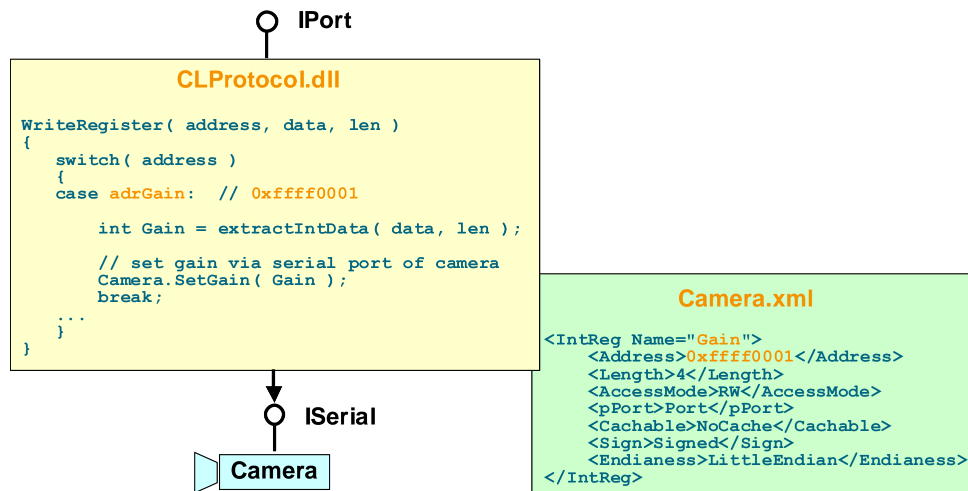


Figure 2 Providing a pseudo register space

This **CLProtocol driver library** has a pure C interface and is normally not used directly. Instead, the **GenICam reference implementation** provides the CLProtocol module which comes with a C++ wrapper class **CCLPort** deals with tasks like loading and binding of the best matching driver library (see Figure 1). The wrapper class

also uses the **CLAllSerial** / **CLSerXXX** mechanism* defined in chapter 3 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 library is not operating system dependent, however the compiled library 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
- Linux 32 bit : this operating system CAN be supported
- Linux 64 bit : this operating system CAN be supported

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

1. The CLProtocol driver libraries and any accompanying XML files must be installed and registered in the system
2. For each frame grabber port the right CLProtocol driver library 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 library are explained. Finally, the properties CLP_DEVICE_BAUDERATE and CLP_DEVICE_SUPPORTED_BAUDERATES are defined which must be implemented by the CLProtocol driver library in order to allow a generic mechanism to connect to setup the baud rate and the camera.

# 2 Installing and Registering CLProtocol Driver Libraries

The CLProtocol driver libraries 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 driver libraries 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 driver library must be installed to. The following sub-directory names are defined:

- For Win32 the sub-directory is **Win32_i86**
- For Win64 the sub-directory is **Win64_x64**
- For Linux 32 bit the sub-directory is **Linux32_i86**
- For Linux 64 bit the sub-directory is **Linux64_x64**

Here is a Windows 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 driver libraries with different names can reside in one sub-directory. The driver library name must be of the form ***.dll** for Windows and **lib*.so** for Linux.

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
```

* 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.

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

If the driver libraries 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 library 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 driver library named XXX.dll/XXX.so should be named XXX.debug.dll/XXX.debug.so. The following rules apply when the driver libraries are enumerated by the CLProtocol module (CCLPort wrapper class):

- If the CLProtocol module is compiled in **Debug** mode a driver library named XXX.dll/XXX.so is loaded only if there is no corresponding driver library named XXX.debug.dll/XXX.debug.so in the same directory.
- If the CLProtocol module is compiled in **Release** mode a driver library named XXX.debug.dll/ XXX.debug.so is loaded only if there is no corresponding driver library named XXX.dll/XXX.so in the same directory.

# 3 Selecting a CLProtocol Driver Library and Identifying a Camera

The key problem when setting up the CLProtocol driver library 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 driver library but it is also required by the CLProtocol driver library itself for adapting its behavior to different camera models of the same vendor.

It would be nice if the manufacturer name as well as the model name of an arbitrary Camera Link camera could be determined automatically just by probing the frame grabber port. However, this kind of plug&play mechanism will stay a dream for Camera Link because for historical reasons there is no standard protocol for the serial port of Camera Link 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 might not recover.

So there is no way around the user selecting at least the camera manufacturer name and thus the CLProtocol driver library for each frame grabber port manually. After that has been done the CLProtocol driver library 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, if for some reason that automation is not possible 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 library. 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 CLProtocol driver library is found (without trailing back slash) and the **file name** of the driver library. 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 manufacturer 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** a 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 an 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 driver libraries installed in the system, each being represented by the corresponding manufacturer name. However it may not be possible for each driver library to fully automatically identify the camera attached to the selected port. For those cases the CLProtocol driver library provides a list of DeviceID templates for the user to select one.

For example the CLProtocol driver library 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 library 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 CLProtocol driver library 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 not bother with automatic identification altogether and just enumerates all camera models the driver library 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 driver libraries 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 library 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 library returns a full DeviceID string unambiguously identifying the camera found connected to the port.

The connection for probing is normally performed at 9600 baud which is by definition of the Camera Link standard the wake up baud rate of cameras. A CLProtocol driver library can optionally implement baud rate auto detection.

Note that any automatic detection of the baud rate typically takes place in the probe step only. When a camera is later re-connected the baud rate is the same are detected in the probe step. The idea is that all time consuming procedures are performed in the probing step and the connection step is made as fast as possible. In order for this to work the CLProtocol driver library must remember the baud rate settings. It can do so because there is a Cookie which is handed out by the probing function and needs to be given to the connect function.

For circumstances where the time consuming probe step needs too much time, e.g. closing the application while probing is active, the CLProtocol driver library may implement the stop probing flag. This flag is called from another thread to immediately abort the current probe procedure (see section 6.2).

**PortIDs**

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

The CLAllSerial module first enumerates all CLSerXXX modules found installed in the system, then it enumerates all frame grabber boards per module 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.

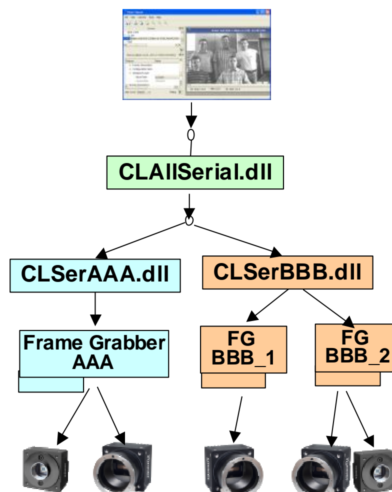


Figure 3 How the CLAllSerial module enumerates frame grabber ports

The enumeration process differs depending on the operating system used. The following search patterns are used when looking for CLSerXXX modules:

- Windows Release Version: `clser???.dll`
- Windows Debug Version: `clser???d.dll`
- Linux Release Version: `clser???.so`
- Linux Debug Version: `clser???d.so`

When CLAllSerial module loads under Windows, it will search for the CLSerXXX modules in the directory of the CLALLSerial module itself and in the following described directory:

For 32-bit and 64-bit Windows the appropriate version of CLSerXXX module should be in the directory defined in the registry key:

```
HKEY_LOCAL_MACHINE\software\cameralink
```

This key contains a value named “CLSERIALPATH” with type string (REG_SZ) which contains the actual path to the directory. The path should be:

```
%ProgramFiles%\CameraLink\Serial
```

If the key/value already exist and point to a different location this location must be used.

For 64-bit Windows the Win32 version of CLSerXXX module should be in the directory defined in the registry key:

```
HKEY_LOCAL_MACHINE\software\Wow6432Node\cameralink
```

*NOTE:CLAllSerial module always uses HKEY_LOCAL_MACHINE\software\cameralink to retrieve the CLSerXXX modules. The Windows Registry Redirector makes sure that the application sees only one of the two registry entries depending on the application being built for Win32 or Win64.*

This key contains a value named “CLSERIALPATH” with type string (REG_SZ) which contains the actual path to the directory. The path should be

```
%ProgramFiles(x86)%\CameraLink\Serial
```

If the key/value already exist and point to a different location this location must be used. You must not change any existing value.

You must not change any existing value. If the keys/values/directories do not exist they must be created.

The value “CLSERIALPATH” must only contain one path.

When CLAllSerial module loads under Linux, it will search for the CLSerXXX modules in the directory of the CLALLSerial module itself. To work with existing installations of frame grabber manufacturers it is recommended to create a soft link in the directory of the CLAllSerial module which points to the real (lib)clserxxx.so laying in the installation path of the frame grabber software.

Example when using it with soft links:

- `/opt/ManufacturerXXX/libclserXXX.so` → CLSerXXX module of ManufacturerXXX
- `~/example/clserXXX.so` → soft link which points to `/opt/ManufacturerXXX/libclserXXX.so`
- `~/example/libCLAllSerial_gcc48_v3_0.so` → CLAllSerial module
- `~/example/libCLProtocol_gcc48_v3_0.so` → CLProtocol module

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 Camera Link 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 library 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 module. This may for example be used in embedded systems. In this case a PortID could for example look like this:

```
"Local#TheOneAndOnlyPort"
```

**Summary**

The following list summarizes the steps a client program has to take in order to select a CLProtocol driver library 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 library 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>@XMLVersion.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 driver library 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 GenICam GenCP standard.
- The CLProtocol driver library itself might contain suitable XML description, e.g. compiled in as Windows resource.
- The directory containing the CLProtocol driver library 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 CLProtocol module of the reference implementation so the CLProtocol driver library 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 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 driver library via the CLAllSerial interface which also provides means to query a list of possible baud rates supported by the grabber (see section 6.2).

GenICam therefore defines the BaudRate as standard feature which must be implemented by the CLProtocol driver library. Besides of the standard baud rates 9600, 19200 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 driver library should implement baud rate **auto detection**, i.e. being able to identify the camera's baud rate during probing.

There must be two ways to access the baud rate.

1. The CLProtocol driver library must implement pseudo registers which are accessible via the IPort interface so the baud rate is exposed via the camera's GenICam XML file.
2. The CLProtocol driver library must also expose the baud rate via the DLL properties which have been introduced in v1.1. This is required so that the CLProtocol module (CCLPort wrapper class) can boost the baud rate while downloading the XML file from the camera.

# 6 Standardized Programming Interfaces

The CLProtocol driver library 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 library
- **CLSerialTypes.h** – declares some types and constants
- **ISerial.h** – declares an abstract C++ interface ISerial which is used by the CLProtocol driver library 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 library 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.

## 6.1 *ISerial Interface*

The CLProtocol driver library needs 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 Camera Link standard.
- **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 Camera Link standard. Because boards supporting only Camera Link v1.0 must be supported no advanced functions like GetNumBytesAvail can be supported.

## 6.2 *CLProtocol Interface*

The functions to be implemented by the CLProtocol driver library are explained along the use cases introduced in the previous sections. Note that the client of the interface described here is a CLProtocol module. The wrapper class **CCLPort** contained in the GenICam reference implementation is an example implementation of a CLProtocol module 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**

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

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 libraries are stored. The CLProtocol module (CCLPort wrapper class) loads each of those driver libraries and calls clpGetShortDeviceIDTemplates retrieving for each driver library a list of short DeviceID templates the respective driver library will understand. The combined short DeviceID templates decorated with the location of the driver library 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 library to distinguish between the two use cases. Re-connecting instead of probing again makes sense because re-connecting is normally much faster than probing. Generally 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 library. The driver library may use the Cookie to persist any data while the connection is open. Note that the Cookie value must not be 0.

#### 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 CLProtocol driver library 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 ID's exposing what kind of XML descriptions the camera and/or the CLProtocol driver library itself is able to provide. Note that this does not include XML descriptions stored on the file system beside the driver library. These XML IDs belonging to these XML files are added by the CLProtocol module (CCLPort wrapper class).

The XML IDs returned from calling clpGetXMLIDs are not sorted. This is also done by the CLProtocol module (CCLPort 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 CLProtocol module (CCLPort wrapper class) handles XML ID's 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 CLProtocol module (CCLPort 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.

#### Event Handling

Is the connected camera and the CLProtocol driver library capable to deal with device events the function **clpGetEventData** is used. The library has to put received device events into an event queue. A call to clpGetEventData delivers the next event from that queue and copies the event data into a user allocated buffer.

The CLProtocol module (CCLPort wrapper class) must provide a check if a CLProtocol driver library supports this functionality.

### Error Handling

Each call to one of the CLProtocol driver library 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 library 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 modules. If the CLProtocol module (CCLPort wrapper class) receives a negative return code it first asks the CLProtocol driver library for an error description text. If that call does not return a valid error message, it calls CLAllSerial module which in turn asks the CLSerXXX module 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 driver library 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 = 1.2.

### Setting and Getting Properties of the CLProtocol driver library (v1.1)

The CLPRotocol driver library itself can have properties which can be get and set by the function pair **clpGetParam** and **clpSetParam**. The properties available are defined by the enumeration **CLP_PARAMS**.

**CLP_LOG_LEVEL** and **CLP_LOG_CALLBACK** are used to set a logging target and a log level. This allows feeding debug messages into GenICam's standard logging system.

**CLP_STOP_PROBE_DEVICE** is used to abort a running probe procedure (v1.2).

**CLP_DEVICE_BAUDERATE** and **CLP_DEVICE_SUPPORTED_BAUDERATES** allow setting the baud rate of the camera as well as the frame grabber (see ISerial interface).

### Initializing the CLProtocol driver library (v1.1)

The functions **clpInitLib** and **clpCloseLib** are called after loading the CLProtocol driver library and shortly before unloading it.