EURESYS™
Excellence in vision

PROGRAMMER GUIDE

Coaxlink

Coaxlink 10.2.1

PCI EXPRESS™

CoaXPress

Terms of Use

EURESYS s.a. shall retain all property rights, title and interest of the documentation of the hardware and the software, and of the trademarks of EURESYS s.a.

All the names of companies and products mentioned in the documentation may be the trademarks of their respective owners.

The licensing, use, leasing, loaning, translation, reproduction, copying or modification of the hardware or the software, brands or documentation of EURESYS s.a. contained in this book, is not allowed without prior notice.

EURESYS s.a. may modify the product specification or change the information given in this documentation at any time, at its discretion, and without prior notice.

EURESYS s.a. shall not be liable for any loss of or damage to revenues, profits, goodwill, data, information systems or other special, incidental, indirect, consequential or punitive damages of any kind arising in connection with the use of the hardware or the software of EURESYS s.a. or resulting of omissions or errors in this documentation.

This documentation is provided with Coaxlink 10.2.1 (doc build 2044).

Contents

1. Introduction

The *Application Programming Interface* (API) for Coaxlink cards is based on GenICam.

The goal of GenICam is to provide a standardized, uniform programming interface for using cameras and frame grabbers based on different physical interfaces (CoaXPress, GigE Vision, etc.) or from different vendors.

GenICam is a set of EMVA standards (GenApi and GenTL), as well as related conventions for naming things (the SFNC for standard features, the PFNC for pixel formats).

- GenApi is about description. At the core of GenApi is the concept of *register description*. Register descriptions are provided in the form of XML files. They map low-level hardware registers to high-level *features*.
  GenApi allows applications to detect, configure and use the features of cameras and frame grabbers in a uniform and consistent way.
- GenTL is about data transport. The TL suffix stands for *Transport Layer*.
  The GenTL standard defines a set of C functions and data types for enumerating, configuring, and grabbing images from cameras and frame grabbers. This API is defined by a C header file.
  Frame grabber vendors provide libraries that implement this API (i.e., libraries that export the functions declared in the standard header file). These libraries are referred to as *GenTL producers*, or *Common Transport Interfaces* (CTI) and use the `cti` file extension. The GenTL producer for Coaxlink cards is `coaxlink.cti`.

This document is meant to be read from beginning to end. Each chapter and section builds upon the preceding ones. If you skip parts of the text, some of the explanations and examples may seem cryptic. If that happens, you should go back and read the parts that you've skipped over.

2. GenApi

GenApi addresses the problem of configuring cameras. The way this is achieved is generic, and applies to different kinds of devices, including frame grabbers. In this chapter, everything we say about cameras also applies to frame grabbers.

GenApi requires two things to work: a *register description*, and a *GenApi implementation*.

Register description

A *register description* is an XML file that can be thought of as a computer-readable datasheet of the camera. It defines camera settings (such as `PixelFormat` and `TriggerSource`), and instructions on how to configure them (e.g., to set `ExposureMode` to `Timed`, write value `0x12` to register `0xE0140`). It can also contain camera documentation.

GenApi implementation

A *GenApi implementation* is a software module that can read and interpret register description files.

The EMVA provides a reference implementation, but it is fairly difficult to use, and logging is very poor. Instead, we recommend using the Euresys implementation bundled with the Coaxlink software package. This implementation also allows writing powerful configuration scripts.

Features

What the user gets from GenApi is a bunch of named *features*, organized in categories.

Set/get features

Set/get features are simple settings (called *parameters* in MultiCam), and can be of different types:

- integer (e.g., `Width`)
- float (e.g., `AcquisitionFrameRate`)
- enumeration (e.g., `PixelFormat`)
- boolean (e.g., `LUTEnable`)
- string (e.g., `DeviceVendorName`)

The value of features can be retrieved/modified using *get/set* functions. Some features are read-only and some are write-only, but most allow read/write access.

Commands

There is also another kind of features: *commands* (e.g., `AcquisitionStart`). Commands are special: they don't have any associated value; they have side effects. Command features are meant to be *executed*. When a command is executed, some action happens in the camera (e.g., a software trigger is generated). Obviously, *get/set* functions don't make sense for commands and can't be used.

3. GenTL

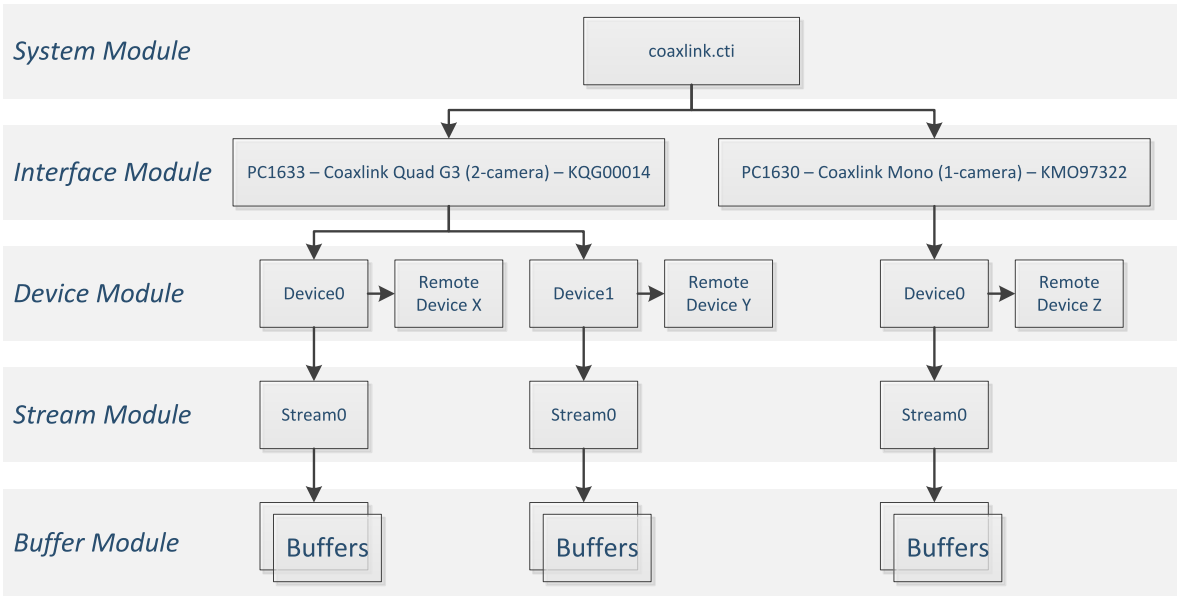
GenTL defines 5 types of objects, organized in a parent/child relationship:

1. the *system* module
2. the *interface* module
3. the *device* module
4. the *data stream* module
5. the *buffer* module

Each module:

- corresponds to a particular element of the system;
- defines relevant pieces of information (*info commands*) that can be queried (using *get info* functions);
- allows exercising that module's functionality (using specific functions).

Additionally, all modules except the buffer module behave as *ports* that allow read/write operations. These port functions are used by GenApi to load that module's description file, and to use its GenApi features.



GenTL Module Hierarchy

3.1. System module

The system module (also referred to as *TLSystem*), represents the GenTL producer (the `coaxlink.cti` library). This module is at the top of the parent/child tree.

The system module provides basic information about the GenTL producer: things like the complete path to `coaxlink.cti` and the vendor name (Euresys).

The real point of the system module is to list the *interfaces* (or frame grabbers) present in the system. The most important functions of the system module are `TLGetNumInterfaces` (to retrieve the number of frame grabbers in the system) and `TLOpenInterface` (to get access to one of the frame grabbers).

3.2. Interface module

The GenTL standard calls frame grabbers *interfaces*. The system module has one child interface for each frame grabber: if there are 2 Coaxlink cards in the computer, the system module will have two child interfaces.

Each interface represents a frame grabber. Global frame grabber features such as digital I/O lines belong in the interface module. This means that the GenApi features controlling the I/O lines are attached to the interface.

Each interface also acts as parent to one or several *devices*. The most important functions of the interface module are `IFGetNumDevices` (to retrieve the number of cameras that can be connected to the interface) and `IFOpenDevice` (to get access to one of the devices).

3.3. Device module

The GenTL standard uses the terms *device* and *remote device* for two related but different concepts. A *remote device* is a real camera, physically connected to a frame grabber. This is different from the device module we describe here.

The device module is the module that contains the frame grabber settings relating to the camera. This includes things like triggers and strobes.

The device module also acts as parent to one *data stream*, and can be viewed as the sibling of the *remote device*. The most important functions of the device module are `DevOpenDataStream` (to get access to the data stream) and `DevGetPort` (to get access to the remote device).

3.4. Data stream module

The data stream module handles *buffers*. During acquisition runs, images are sent from the camera to the frame grabber, which transfers them to memory buffers allocated on the host computer. The data stream module is where image acquisition occurs. It is where most of the functionality resides.

Buffer handling is very flexible. Any number of buffers can be used. Buffers are either in the input queue, in the output queue, or temporarily unqueued. The application decides when empty buffers are queued (to the input FIFO), and when filled buffers are popped (from the output FIFO).

3.5. Buffer module

The buffer module simply represents a memory buffer given to a parent data stream. Useful metadata is associated to buffers. This includes the image width, height, pixel format, timestamp... These are retrieved through *info commands* (see `BUFFER_INFO_CMD_LIST` in the standard GenTL header file.

The buffer module is the only module that doesn't have read/write port functions; it doesn't have GenApi features.

3.6. GenTL API

GenTL makes it possible to detect, control and use all camera and frame grabber features, but its usage is tedious:

- `cti` files must be dynamically loaded, and the functions they export must be accessed through pointers.
- Functions return an error code that must be checked by the application.
- Most functions read from/write to untyped buffers: the application must determine the required buffer size, allocate a temporary buffer, convert data to/from this buffer, and finally release the buffer memory.

Instead of using the GenTL API directly, we recommend using either:

- the `Euresys::EGenTL` library which deals with these complications so that the user doesn't have to;
- or the `Euresys::EGrabber` library which provides a high-level, easy-to-use interface.

4. Euresys::EGenTL

`Euresys::EGenTL` is a library of C++ classes that provide the same functionality as standard GenICam GenTL, but with a more user-friendly interface. For example, it uses `std::string` instead of raw `char` pointers, and error codes are transformed into exceptions. `Euresys::EGenTL` also takes care of locating the GenTL producer and loading the functions it exports.

This library is implemented entirely in C++ header files. As a result, you can simply include the relevant header file:

```
#include <EGenTL.h>
```

Instead of the raw, low-level C functions that GenTL defines, we get a `Euresys::EGenTL` object that represents the GenTL producer:

- Each GenTL function is available as a member method of `Euresys::EGenTL`. GenTL function names start with an upper-case prefix. In `Euresys::EGenTL`, method names start with the same prefix, but written in lower-case. For example, the `GCReadPort` function is exposed as the `gcReadPort` method, and the `TLOpenInterface` function as the `tlOpenInterface` method.
- All GenTL functions return a `GC_ERROR` code indicating success or failure. When a function returns a code other than `GC_ERR_SUCCESS`, an exception is thrown. This removes the burden of manually checking error codes after each function call.
- Since GenTL functions return a `GC_ERROR`, output values are returned through pointers passed as arguments. `Euresys::EGenTL` methods offer a more natural interface; they return the output value directly:

```
GC_API TLGetNumInterfaces(TL_HANDLE hTL, uint32_t *piNumIfaces);
```

```
uint32_t tlGetNumInterfaces(TL_HANDLE tlh);
```

> **Note:** *`GC_API` is defined as `GC_IMPORT_EXPORT GC_ERROR GC_CALLTYPE`. It is simply a `GC_ERROR` decorated with calling convention and DLL import/export attributes.*

- For GenTL functions that deal with text, the corresponding `Euresys::EGenTL` methods convert from `char *` to `std::string` and vice-versa:

```
GC_API TLGetInterfaceID(TL_HANDLE hTL, uint32_t iIndex,
                        char *sID, size_t *piSize);
```

```
std::string tlGetInterfaceID(TL_HANDLE tlh, uint32_t index);
```

- Some GenTL functions retrieve information about the camera or frame grabber. These functions fill a `void *` buffer with a value, and indicate in an `INFO_DATATYPE` the actual type of the value. `Euresys::EGenTL` uses C++ templates to make these functions easy to use:

```
GC_API GCGetInfo(TL_INFO_CMD iInfoCmd, INFO_DATATYPE *piType,
                 void *pBuffer, size_t *piSize);
```

```
template<typename T> T gcGetInfo(TL_INFO_CMD cmd);
```

4.1. A first example

This program uses `Euresys::EGenTL` to iterate over the Coaxlink cards present in the system, and display their id:

```
#include <iostream>
#include <EGenTL.h>                                          // 1

void listCards() {
    Euresys::EGenTL gentl;                                   // 2
    GenTL::TL_HANDLE tl = gentl.tlOpen();                    // 3
    uint32_t numCards = gentl.tlGetNumInterfaces(tl);        // 4
    for (uint32_t n = 0; n < numCards; ++n) {
        std::string id = gentl.tlGetInterfaceID(tl, n);      // 5
        std::cout << "[" << n << "] " << id << std::endl;
    }
}

int main() {
    try {                                                    // 6
        listCards();
    } catch (const std::exception &e) {                      // 6
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

1. Include `EGenTL.h`, which contains the definition of the `Euresys::EGenTL` class.
2. Create a `Euresys::EGenTL` object. This involves the following operations:
   - locate and dynamically load the Coaxlink GenTL producer (`coaxlink.cti`);
   - retrieve pointers to the functions exported by `coaxlink.cti`, and make them available via `Euresys::EGenTL` methods;
   - initialize `coaxlink.cti` (this is done by calling the GenTL initialization function `GCInitLib`).
3. Open the GenTL producer. This returns a handle of type `GenTL::TL_HANDLE`. The `GenTL` namespace is defined in the standard GenTL header file, which has been automatically included by `EGenTL.h` in step 1.
4. Find out how many cards are present in the system.
5. Retrieve the id of the n-th card.
6. `Euresys::EGenTL` uses exceptions to report errors, so we wrap our code inside a `try ... catch` block.

Example of program output

```
[0] PC1633 - Coaxlink Quad G3 (1-camera, line-scan) - KQG00014
[1] PC1632 - Coaxlink Quad (1-camera) - KQU00031
```

4.2. Relevant files

`include/EGenTL.h`	Main header. Includes all the other headers. Defines `Euresys::EGenTL`
`include/GenTL_v1_5.h`	Standard GenTL header. Defines standard types, functions and constants.
`include/GenTL_v1_5_EuresysCustom.h`	Defines Coaxlink-specific constants

5. Euresys::EGrabber

`Euresys::EGrabber` is a library of C++ classes that provide a high-level interface. It is built on top of the `Euresys::EGenTL` library, and is recommended for most users.

A .NET assembly, built on top of the `Euresys::EGrabber` C++ classes, is also provided. In this document, we focus mainly on the C++ API. Minor differences between the C++ and .NET interfaces are listed in a dedicated chapter.

To use the classes described here, you need to include the main `Euresys::EGrabber` file:

```
#include <EGrabber.h>
```

`Euresys::EGrabber` is a header-only library (it isn't provided as a `lib` or `dll` file). It comprises several classes, the most important of which is also named `Euresys::EGrabber`:

```
namespace Euresys {
    class EGrabber;
}
```

In this text, we'll refer to this class as a *grabber*. A grabber encapsulates a set of related GenTL modules:

- An interface: the module that represents global (shared) frame grabber settings and features. This includes digital I/O control, PCIe and firmware status...
- A device (or *local* device, as opposed to *remote* device): the module that contains the frame grabber settings and features relating to the camera. This consists mainly of camera and illumination control features: strobes, triggers...
- A data stream: the module that handles image buffers.
- A remote device: the CoaXPress camera.
- A number of buffers.

Go back to the chapter about GenTL modules if these concepts are not clear.

5.1. A first example

This example creates a grabber and displays basic information about the interface, device, and remote device modules it contains:

```
#include <iostream>
#include <EGrabber.h>                                                                   // 1

static const uint32_t CARD_IX = 0;
static const uint32_t DEVICE_IX = 0;

void showInfo() {
    Euresys::EGenTL gentl;                                                              // 2
    Euresys::EGrabber<> grabber(gentl, CARD_IX, DEVICE_IX);                             // 3

    std::string card = grabber.getString<Euresys::InterfaceModule>("InterfaceID");      // 4
    std::string dev = grabber.getString<Euresys::DeviceModule>("DeviceID");             // 5
    int64_t width = grabber.getInteger<Euresys::RemoteModule>("Width");                 // 6
    int64_t height = grabber.getInteger<Euresys::RemoteModule>("Height");               // 6

    std::cout << "Interface:    " << card << std::endl;
    std::cout << "Device:       " << dev  << std::endl;
    std::cout << "Resolution:   " << width << "x" << height << std::endl;
}

int main() {
    try {                                                                               // 7
        showInfo();
    } catch (const std::exception &e) {                                                 // 7
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

1. Include `EGrabber.h`, which defines the `Euresys::EGrabber` class, and includes the other header files we need (such as `EGenTL.h` and the standard GenTL header file).
2. Create a `Euresys::EGenTL` object. This involves the following operations:
   - locate and dynamically load the Coaxlink GenTL producer (`coaxlink.cti`);
   - retrieve pointers to the functions exported by `coaxlink.cti`, and make them available via `Euresys::EGenTL` methods;
   - initialize `coaxlink.cti` (this is done by calling the GenTL initialization function `GCInitLib`).
3. Create a `Euresys::EGrabber` object. The constructor needs the `gentl` object created in step 2. It also takes as optional arguments the indices of the interface and device to use. The purpose of the angle brackets (<>) that come after `EGrabber` will become clear later. For now, they can be safely ignored.
4. Use "GenApi" on page 6 to find out the ID of the Coaxlink card. `Euresys::InterfaceModule` indicates that we want an answer from the interface module.

- Similarly, find out the ID of the device. This time, we use `Euresys::DeviceModule` to target the device module.
- Finally, read the camera resolution. `Euresys::RemoteModule` indicates that the value must be retrieved from the camera.

- `Euresys::EGrabber` uses exceptions to report errors, so we wrap our code inside a `try ... catch` block.

Example of program output

```
Interface:    PC1633 - Coaxlink Quad G3 (2-camera) - KQG00014
Device:       Device0
Resolution:   4096x4096
```

5.2. Acquiring images

This program uses `Euresys::EGrabber` to acquire images from a camera connected to a Coaxlink card:

```
#include <iostream>
#include <EGrabber.h>

void grab() {
    Euresys::EGenTL gentl;
    Euresys::EGrabber<> grabber(gentl);                                          // 1

    grabber.reallocBuffers(3);                                                   // 2
    grabber.start(10);                                                           // 3
    for (size_t i = 0; i < 10; ++i) {
        Euresys::ScopedBuffer buf(grabber);                                      // 4
        void *ptr = buf.getInfo<void *>(GenTL::BUFFER_INFO_BASE);                // 5
        uint64_t ts = buf.getInfo<uint64_t>(GenTL::BUFFER_INFO_TIMESTAMP);       // 6
        std::cout << "buffer address: " << ptr << ", timestamp: "
                  << ts << " us" << std::endl;
    }                                                                            // 7
}

int main() {
    try {
        grab();
    } catch (const std::exception &e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

1. Create a `Euresys::EGrabber` object. The second and third arguments of the constructor are omitted here. The grabber will use the first device of the first interface present in the system.
2. Allocate 3 buffers. The grabber automatically determines the required buffer size.
3. Start the grabber. Here, we ask the grabber to fill 10 buffers. If we don't want the grabber to stop after a specific number of buffers, we can do `grabber.start(GenTL::GENTL_INFINITE)`, or simply `grabber.start()`.
   Starting the grabber involves the following operations:
   - the `AcquisitionStart` command is executed on the camera;
   - the `DSStartAcquisition` function is called to start the data stream.

   In this example, we assume that the camera and frame grabber are properly configured. For a real application, it would be safer to run a configuration script before starting acquisitions (and before allocating buffers for that matter). This will be shown in another example.

4. Wait for a buffer filled by the grabber. The result is a `Euresys::ScopedBuffer`. The term *scoped* is used to indicate that the lifetime of the buffer is the current scope (i.e., the current block).

5. Retrieve the buffer address. This is done by calling the `getInfo` method of the buffer. This method takes as argument a `BUFFER_INFO_CMD`. In this case, we request the `BUFFER_INFO_BASE`, which is defined in the standard GenTL header file:

```
enum BUFFER_INFO_CMD_LIST
{
  BUFFER_INFO_BASE      = 0, /* PTR    Base address of the buffer memory. */
  BUFFER_INFO_SIZE      = 1, /* SIZET  Size of the buffer in bytes. */
  BUFFER_INFO_USER_PTR  = 2, /* PTR    Private data pointer of the GenTL Consumer. */
  BUFFER_INFO_TIMESTAMP = 3, /* UINT64 Timestamp the buffer was acquired. */
  // ...
  // other BUFFER_INFO definitions omitted
  // ...
  BUFFER_INFO_CUSTOM_ID = 1000 /* Starting value for GenTL Producer custom IDs. */
};
typedef int32_t BUFFER_INFO_CMD;
```

   Notice that `getInfo` is a template method, and when we call it we must specify the type of value we expect. `BUFFER_INFO_BASE` returns a pointer; this is why we use `getInfo<void *>`.

6. Do the same to retrieve the timestamp of the buffer. This time, we use the `uint64_t` version of `getInfo` to match the type of `BUFFER_INFO_TIMESTAMP`.

   Note: *For Coaxlink, timestamps are always 64-bit and expressed as the number of microseconds that have elapsed since the computer was started.*

7. We reach the end of the `for` block. The local variable `buf` gets out of scope and is destroyed: the `ScopedBuffer` destructor is called. This causes the GenTL buffer contained in `buf` to be re-queued (given back) to the data stream of the grabber.

Example of program output

```
buffer address: 0x7f3c32c54010, timestamp: 11247531003686 us
buffer address: 0x7f3c2c4bf010, timestamp: 11247531058080 us
buffer address: 0x7f3c2c37e010, timestamp: 11247531085003 us
buffer address: 0x7f3c32c54010, timestamp: 11247531111944 us
buffer address: 0x7f3c2c4bf010, timestamp: 11247531137956 us
buffer address: 0x7f3c2c37e010, timestamp: 11247531163306 us
buffer address: 0x7f3c32c54010, timestamp: 11247531188600 us
buffer address: 0x7f3c2c4bf010, timestamp: 11247531213807 us
buffer address: 0x7f3c2c37e010, timestamp: 11247531239158 us
buffer address: 0x7f3c32c54010, timestamp: 11247531265053 us
```

We can see that the three buffers that were allocated (let's call them A at `0x7f3c32c54010`, B at `0x7f3c2c4bf010`, and C at `0x7f3c2c37e010`) are used in a round-robin fashion: A → B → C → A → B → C → ... This is the result of:

- the FIFO nature of input and output buffer queues:
  - the Coaxlink driver pops a buffer from the front of the input queue, and gives it to the Coaxlink card for DMA transfer;
  - when the transfer is complete, the buffer is pushed to the back of the output queue;
- the use of `ScopedBuffer`:

- the `ScopedBuffer` constructor pops a buffer from the front of the output queue (i.e., it takes the oldest buffer);
- the `ScopedBuffer` destructor pushes that buffer to the back of the input queue (hence, this buffer will be used for a new transfer after all buffers already in the input queue).

5.3. Configuring the grabber

Configuration is a very important aspect of any image acquisition program.

- The camera and the frame grabber both have to be configured according to the application requirements.
- The camera configuration must be compatible with the frame grabber configuration, and vice versa.

Configuration basically boils down to a series of set/get operations performed on the grabber modules: the remote device (i.e., the camera), the interface, the device, or the data stream modules.

This program configures the grabber for the so-called *RG* control mode (asynchronous reset camera control, frame grabber-controlled exposure).

```
#include <iostream>
#include <EGrabber.h>

const double FPS = 150;

void configure() {
    Euresys::EGenTL gentl;
    Euresys::EGrabber<> grabber(gentl);
    // camera configuration
    grabber.setString<Euresys::RemoteModule>("TriggerMode", "On");                      // 1
    grabber.setString<Euresys::RemoteModule>("TriggerSource", "CXPin");                 // 2
    grabber.setString<Euresys::RemoteModule>("ExposureMode", "TriggerWidth");           // 3
    // frame grabber configuration
    grabber.setString<Euresys::DeviceModule>("CameraControlMethod", "RG");              // 4
    grabber.setString<Euresys::DeviceModule>("CycleTriggerSource", "Immediate");        // 5
    grabber.setFloat<Euresys::DeviceModule>("CycleMinimumPeriod", 1e6 / FPS);           // 6
}

int main() {
    try {
        configure();
    } catch (const std::exception &e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

1. Enable triggers on the camera.
2. Tell the camera to look for triggers on the CoaXPress link.
3. Configure the camera to use the `TriggerWidth` exposure mode.
4. Set the frame grabber's camera control method to `RG`. In this mode, camera cycles are initiated by the frame grabber, and the exposure duration is also controlled by the frame grabber.

5. Tell the frame grabber to initiate camera cycles itself (at a rate defined by `CycleMinimumPeriod`), without waiting for hardware or software triggers.
6. Configure the frame rate.

But there is a better way to configure the grabber. Using a script file, the program becomes:

```
#include <iostream>
#include <EGrabber.h>

void configure() {
    Euresys::EGenTL gentl;
    Euresys::EGrabber<> grabber(gentl);
    grabber.runScript("config.js");
}

int main() {
    try {
        configure();
    } catch (const std::exception &e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

and the configuration script is:

```
var grabber = grabbers[0];
var FPS = 150;
// camera configuration
grabber.RemotePort.set("TriggerMode", "On");
grabber.RemotePort.set("TriggerSource", "CXPin");
grabber.RemotePort.set("ExposureMode", "TriggerWidth");
// frame grabber configuration
grabber.DevicePort.set("CameraControlMethod", "RG");
grabber.DevicePort.set("CycleTriggerSource", "Immediate");
grabber.DevicePort.set("CycleMinimumPeriod", 1e6 / FPS);
```

Using a script file has several advantages:

- The configuration can be changed without recompiling the application. This allows shorter development cycles, and makes it possible to update the configuration in the lab or in the field.
- The configuration script can be loaded by the GenICam Browser and the command-line `gentl` tool. This makes is possible to validate the configuration outside of the user application.
- The configuration script can easily be shared by several applications written in different programming languages: C++, C#, VB.NET...
- The full power of Euresys GenApi scripts is available.

5.4. Events

Background

Coaxlink cards generate different kinds of events:

- *New buffer* events: events indicating that a buffer has been filled by a data stream.

- *Data stream* events: events related to a data stream and its frame store.
- *Camera and illumination controller* events: events related to the real-time control (performed by a device) of a camera and its illumination devices.
- *I/O toolbox* events: events (coming from the interface) related to digital I/O lines and other I/O tools.
- *CoaXPress interface* events: events (also coming from the interface) related to the CoaXPress interface.

New buffer events are standard in GenTL. They occur when a buffer is filled by the frame grabber. Information attached to *new buffer* events include the handle of the buffer and a timestamp.

The other types of events are restricted to Coaxlink and can be viewed as categories of specific events. For example, in the *CIC* category of events, we have:

- `CameraTriggerRisingEdge` (start of camera trigger)
- `CameraTriggerFallingEdge` (end of camera trigger)
- `StrobeRisingEdge` (start of light strobe)
- `StrobeFallingEdge` (end of light strobe)
- `AllowNextCycle` (CIC is ready for next camera cycle)
- ...

and in the *I/O toolbox* category of events, we have:

- `LIN1` (line input tool 1)
- `LIN2` (line input tool 2)
- `MDV1` (multiplier/divider tool 1)
- ...

Counters

Coaxlink firmware counts each occurrence of each event (except *new buffer* events) and makes this counter available in a GenApi feature named `EventCount`. Each event has its own counter, and the value of `EventCount` depends on the selected event:

```
// select the CameraTriggerRisingEdge event
grabber.setString<DeviceModule>("EventSelector", "CameraTriggerRisingEdge");
// read the value of the counter
int64_t counter = grabber.getInteger<DeviceModule>("EventCount");
```

or, using the *selected feature* notation:

```
// read the value of the CameraTriggerRisingEdge counter
int64_t counter = grabber.getInteger<DeviceModule>("EventCount[CameraTriggerRisingEdge]");
```

Notifications

As we've just seen, when an event occurs, a dedicated counter is incremented. Coaxlink can also notify the application of this event by having `Euresys::EGrabber` execute a user-defined callback function. But first, it is required to enable notifications of one or more events:

```
grabber.setString<DeviceModule>("EventSelector", "CameraTriggerRisingEdge");
grabber.setInteger<DeviceModule>("EventNotification", true);
grabber.setString<DeviceModule>("EventSelector", "CameraTriggerFallingEdge");
grabber.setInteger<DeviceModule>("EventNotification", true);
...
```

or:

```
grabber.setInteger<DeviceModule>("EventNotification[CameraTriggerRisingEdge]", true);
grabber.setInteger<DeviceModule>("EventNotification[CameraTriggerFallingEdge]", true);
...
```

Using a configuration script, it is easy to enable notifications for all events:

```
function enableAllEvents(p) {                          // 1
    var events = p.$ee('EventSelector');               // 2
    for (var e of events) {
        p.set('EventNotification[' + e + ']', true);   // 3
    }
}

var grabber = grabbers[0];
enableAllEvents(grabber.InterfacePort);                // 4
enableAllEvents(grabber.DevicePort);                   // 5
enableAllEvents(grabber.StreamPort);                   // 6
```

1. Define a helper function named `enableAllEvents` and taking as argument a module (or port) `p`.
2. Use the `$ee` function to retrieve the list of values `EventSelector` can take. This is the list of events generated by module `p`. (`ee` stands for *enum entry*.)
3. For each event, enable notifications. (The `+` operator concatenates strings, so if `e` is `'LIN1'`, the expression `'EventNotification[' + e + ']'` evaluates to `'EventNotification[LIN1]'`.)
4. Call the `enableAllEvents` function defined in step 1 for the interface module. This will enable notifications for all events in the *I/O toolbox* and *CoaXPress interface* categories.
5. Likewise, enable notifications for all events coming from the device module (*CIC* events).
6. Finally, enable notifications for all *data stream* events.

Callback functions

When an event occurs, and event notification is enabled for that event, `Euresys::EGrabber` executes one of several callback functions.

These callback functions are defined in overridden virtual methods:

```
class MyGrabber : public Euresys::EGrabber<>
{
    public:
        ...
```

```
    private:
        // callback function for new buffer events
        virtual void onNewBufferEvent(const NewBufferData& data) {
            ...
        }

        // callback function for data stream events
        virtual void onDataStreamEvent(const DataStreamData &data) {
            ...
        }

        // callback function for CIC events
        virtual void onCicEvent(const CicData &data) {
            ...
        }

        // callback function for I/O toolbox events
        virtual void onIoToolboxEvent(const IoToolboxData &data) {
            ...
        }

        // callback function for CoaXPress interface events
        virtual void onCxpInterfaceEvent(const CxpInterfaceData &data) {
            ...
        }
};
```

As you can see, a different callback function can be defined for each category of events.

In .NET, callback functions are defined by creating delegates rather than overriding virtual methods. An example will be given in the chapter about the .NET assembly.

Event identification

When an event is notified to the application, the callback function that is executed indicates the category of that event. The actual event that occurred is identified by a numerical ID, called `numid`, and defined in `include/GenTL_v1_5_EuresysCustom.h`:

```
enum EVENT_DATA_NUMID_CUSTOM_LIST
{
    // EVENT_CUSTOM_IO_TOOLBOX
    EVENT_DATA_NUMID_IO_TOOLBOX_LIN1                    = ...  /* Line Input Tool 1 */
    EVENT_DATA_NUMID_IO_TOOLBOX_LIN2                    = ...  /* Line Input Tool 2 */
    EVENT_DATA_NUMID_IO_TOOLBOX_MDV1                    = ...  /* Multiplier/Divider Tool 1 */
    ...
    // EVENT_CUSTOM_CXP_INTERFACE
    ...
    // EVENT_CUSTOM_CIC
    EVENT_DATA_NUMID_CIC_CAMERA_TRIGGER_RISING_EDGE     = ...  /* Start of camera trigger */
    EVENT_DATA_NUMID_CIC_CAMERA_TRIGGER_FALLING_EDGE    = ...  /* End of camera trigger */
    EVENT_DATA_NUMID_CIC_STROBE_RISING_EDGE             = ...  /* Start of light strobe */
    EVENT_DATA_NUMID_CIC_STROBE_FALLING_EDGE            = ...  /* End of light strobe */
    ...
    // EVENT_CUSTOM_DATASTREAM
    EVENT_DATA_NUMID_DATASTREAM_START_OF_CAMERA_READOUT = ...  /* Start of camera readout */
    EVENT_DATA_NUMID_DATASTREAM_END_OF_CAMERA_READOUT   = ...  /* End of camera readout */
    ...
};
```

For reference, the following tables list, for each module generating events and for each category of events, the relationships with:

- the name of the callback function

- the data type passed to the callback function
- the common `numid` prefix

Note: *A simple naming scheme is followed: a category of events named some category has a callback function named `on`**`SomeCategory`**`Event` which takes as argument a **`SomeCategory`**`Data` structure, and uses `EVENT_DATA_NUMID_`**`SOME_CATEGORY_`** as common `numid` prefix.*

Data stream module - New Buffer category

Callback function	Data type	`numid` prefix
onNewBufferEvent	NewBufferData	-

Note: *There is only one event in the new buffer event category, so we don't need a `numid` there.*

Data stream module - Data Stream category

Callback function	Data type	`numid` prefix
onDataStreamEvent	DataStreamData	EVENT_DATA_NUMID_DATASTREAM_

Device module - CIC category

Callback function	Data type	`numid` prefix
onCicEvent	CicData	EVENT_DATA_NUMID_CIC_

Interface module - I/O Toolbox category

Callback function	Data type	`numid` prefix
onIoToolboxEvent	IoToolboxData	EVENT_DATA_NUMID_IO_TOOLBOX_

Interface module - CXP Interface category

Callback function	Data type	`numid` prefix
onCxpInterfaceEvent	CxpInterfaceData	EVENT_DATA_NUMID_CXP_INTERFACE_

Examples

We'll soon show a few complete example programs illustrating events and callbacks, but there is one more thing we need to explain before we can do that: the context in which callback functions are executed. This is the subject of the next section.

5.5. EGrabber flavors

When should the callback functions be called? From which context (i.e., which thread)? Thinking about these questions leads to the definition of several *callback models*:

- The application asks the grabber: *"Do you have any buffer or event for me? If yes, execute my callback function now."* This is a polling, synchronous mode of operation, where callbacks are executed when the application demands it, in the application thread.
  We'll refer to this callback model as *on demand*.
- The application asks the grabber to create a single, dedicated thread, and to wait for events in this thread. When an event occurs, the grabber executes the corresponding callback function, also in this single callback thread.
  We'll refer to this callback model as *single thread*.
- The application asks the grabber to create a dedicated thread for each of its callback functions. In each of these threads, the grabber waits for a particular category of events. When an event occurs, the corresponding callback function is executed in that thread.
  We'll refer to this callback model as *multi thread*.

These three callback models all make sense, and each one is best suited for some applications.

- The *on demand* model is the simplest. Although its implementation may use worker threads, from the point of view of the user it doesn't add any thread. This means that the application doesn't need to worry about things such as thread synchronization, mutexes, etc.
- If the user wants a dedicated callback thread, the *single thread* model creates it for him.When we have only one thread, things are simple. In this model, the grabber is used in (at least) two threads, so we need to start worrying about synchronization and shared data.
- In the *multi thread* model, each category of event gets its own thread. The benefit of this is that events of one type (and the execution of their callback functions) don't delay notifications of events of other types. For example, a thread doing heavy image processing in `onNewBufferEvent` will not delay the notification of CIC events by `onCicEvent` (e.g., events indicating that the exposure is complete and that the object or camera can be moved).
  In this model, the grabber is used in several thread, so the need for synchronization is present as it is in the *single thread* model.
  Of course, the application must also be aware that it might receive notifications for events of type *X* that are older than notifications of events of type *Y* that have already been received and processed. After all, this is the differentiating factor between the *single thread* and *multi thread* models.

To give the user maximum flexibility, we support all three callback models. This is why `Euresys::EGrabber` exists in different flavors. So far, we have eluded the meaning of the angle brackets in `EGrabber<>`. The `EGrabber` class is actually a *template class*, i.e., a class that is parameterized by another type:

- In this case, the template parameter is the callback model to use: one of `CallbackOnDemand`, `CallbackSingleThread` or `CallbackMultiThread`.
- The empty `<>` are used to select the default template parameter, which is `CallbackOnDemand`.

- The types of grabber that can be instantiated are:
  - `EGrabber<CallbackOnDemand>`
  - `EGrabber<CallbackSingleThread>`
  - `EGrabber<CallbackMultiThread>`
  - `EGrabber<>` which is equivalent to `EGrabber<CallbackOnDemand>`
- The `EGrabber` header file also defines the following type aliases (synonyms):

```
typedef EGrabber<CallbackOnDemand>     EGrabberCallbackOnDemand;
typedef EGrabber<CallbackSingleThread> EGrabberCallbackSingleThread;
typedef EGrabber<CallbackMultiThread>  EGrabberCallbackMultiThread;
```

- The .NET generics don't quite match the C++ templates, so in .NET the template `EGrabber` class does not exist and we must use one of `EGrabberCallbackOnDemand`, `EGrabberCallbackSingleThread` or `EGrabberCallbackMultiThread`.

5.6. Events and callbacks examples

On demand callbacks

This program displays basic information about CIC events generated by a grabber:

```
#include <iostream>
#include <EGrabber.h>

using namespace Euresys;                                                    // 1

class MyGrabber : public EGrabber<CallbackOnDemand> {                       // 2
    public:
        MyGrabber(EGenTL &gentl) : EGrabber<CallbackOnDemand>(gentl) {      // 3
            runScript("config.js");                                         // 4
            enableEvent<CicData>();                                         // 5
            reallocBuffers(3);
            start();
        }

    private:
        virtual void onCicEvent(const CicData &data) {
            std::cout << "timestamp: " << std::dec << data.timestamp << " us, " // 6
                      << "numid: 0x" << std::hex << data.numid              // 6
                      << " (" << getEventDescription(data.numid) << ")"
                      << std::endl;
        }
};

int main() {
    try {
        EGenTL gentl;
        MyGrabber grabber(gentl);
        while (true) {
            grabber.processEvent<CicData>(1000);                            // 7
        }
    } catch (const std::exception &e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

1. This *using* directive allows writing `Xyz` instead of `Euresys::Xyz`. This helps keep lines relatively short.
2. Define a new class `MyGrabber` which is derived from `EGrabber<CallbackOnDemand>`.
3. `MyGrabber`'s constructor initializes its base class by calling `EGrabber<CallbackOnDemand>`'s constructor.
4. Run a `config.js` script which should:
   - properly configure the camera and frame grabber;
   - enable notifications for CIC events.
5. Enable `onCicEvent` callbacks.
6. The `onCicEvent` callback function receives a `const CicData &`. This structure is defined in `include/EGrabberTypes.h`. It contains a few pieces of information about the event that occurred. Here, we display the `timestamp` and `numid` of each event. The `numid` indicates which CIC event occurred.
7. Call `processEvent<CicData>(1000)`:
   - the grabber starts waiting for a CIC event;
   - if an event occurs within `1000` ms, the grabber executes the `onCicEvent` callback function;
   - otherwise, a *timeout* exception will be thrown.

Example of program output:

```
timestamp: 1502091779 us, numid: 0x8041 (Start of camera trigger)
timestamp: 1502091784 us, numid: 0x8048 (Received acknowledgement for previous CXP trigger message)
timestamp: 1502091879 us, numid: 0x8043 (Start of light strobe)
timestamp: 1502092879 us, numid: 0x8044 (End of light strobe)
timestamp: 1502097279 us, numid: 0x8042 (End of camera trigger)
timestamp: 1502097284 us, numid: 0x8048 (Received acknowledgement for previous CXP trigger message)
timestamp: 1502191783 us, numid: 0x8041 (Start of camera trigger)
timestamp: 1502191783 us, numid: 0x8045 (CIC is ready for next camera cycle)
timestamp: 1502191788 us, numid: 0x8048 (Received acknowledgement for previous CXP trigger message)
timestamp: 1502191883 us, numid: 0x8043 (Start of light strobe)
timestamp: 1502192883 us, numid: 0x8044 (End of light strobe)
timestamp: 1502197283 us, numid: 0x8042 (End of camera trigger)
timestamp: 1502197288 us, numid: 0x8048 (Received acknowledgement for previous CXP trigger message)
timestamp: 1502291788 us, numid: 0x8041 (Start of camera trigger)
timestamp: 1502291788 us, numid: 0x8045 (CIC is ready for next camera cycle)
...
```

Single thread and multi thread callbacks

This program displays basic information about CIC events generated by a grabber, this time using the `CallbackSingleThread` model.

```
#include <iostream>
#include <EGrabber.h>

using namespace Euresys;

class MyGrabber : public EGrabber<CallbackSingleThread> {                       // 1
    public:
        MyGrabber(EGenTL &gentl) : EGrabber<CallbackSingleThread>(gentl) {      // 2
            runScript("config.js");
            enableEvent<CicData>();
            reallocBuffers(3);
            start();
        }

    private:
        virtual void onCicEvent(const CicData &data) {
            std::cout << "timestamp: " << std::dec << data.timestamp << " us, "
                      << "numid: 0x" << std::hex << data.numid
                      << " (" << getEventDescription(data.numid) << ")"
                      << std::endl;
        }
};

int main() {
    try {
        EGenTL gentl;
        MyGrabber grabber(gentl);
        while (true) {                                                          // 3
        }
    } catch (const std::exception &e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

There are very few differences between this program and the `CallbackOnDemand` version:

1. `MyGrabber` is derived from `EGrabber<CallbackSingleThread>` instead of `EGrabber<CallbackOnDemand>`.
2. Consequently, `MyGrabber`'s constructor initializes its base class by calling `EGrabber<CallbackSingleThread>`'s constructor.
3. `EGrabber` creates a callback thread in which it calls `processEvent`, so we don't have to. Here, we simply enter an infinite loop.

As you can see, moving from `CallbackOnDemand` to `CallbackSingleThread` is very simple. If instead you want the `CallbackMultiThread` variant, simply change the base class of `MyGrabber` to `EGrabber<CallbackMultiThread>` (and call the appropriate constructor).

New buffer callbacks

This program shows how to access information related to *new buffer* events. It uses `CallbackMultiThread`, but it could use another callback method just as well.

```
#include <iostream>
#include <EGrabber.h>

using namespace Euresys;

class MyGrabber : public EGrabber<CallbackMultiThread> {
    public:
        MyGrabber(EGenTL &gentl) : EGrabber<CallbackMultiThread>(gentl) {
            runScript("config.js");
            enableEvent<NewBufferData>();
            reallocBuffers(3);
            start();
        }

    private:
        virtual void onNewBufferEvent(const NewBufferData &data) {
            ScopedBuffer buf(*this, data);                                      // 1
            uint64_t ts = buf.getInfo<uint64_t>(GenTL::BUFFER_INFO_TIMESTAMP);  // 2
            std::cout << "event timestamp: " << data.timestamp << " us, "       // 3
                      << "buffer timestamp: " << ts << " us" << std::endl;
        }                                                                       // 4
};

int main() {
    try {
        EGenTL gentl;
        MyGrabber grabber(gentl);
        while (true) {
        }
    } catch (const std::exception &e) {
        std::cout << "error: " << e.what() << std::endl;
    }
}
```

1. In `onNewBufferEvent`, create a temporary `ScopedBuffer` object `buf`. The `ScopedBuffer` constructor takes two arguments:
   - the grabber owning the buffer: since we are in a class derived from `EGrabber`, we simply pass `*this`;
   - information about the buffer: this is provided in `data`.
2. Retrieve the timestamp of the buffer, which is defined as the time at which *the camera started to send data to the frame grabber*.
3. As explained in the section about event identification, *new buffer* events are slightly different from the other kinds of events: they are standard (as per GenTL), and don't have an associated `numid`.
   As a consequence, the `NewBufferData` structure passed to `onNewBufferEvent` doesn't have a `numid` field. It does, however, have a `timestamp` field indicating the time at which *the driver was notified that data transfer to the buffer was complete*. This *event timestamp* is inevitably greater than the *buffer timestamp* retrieved in step 2.
4. We reach the end of the block where the local variable `buf` has been created. It gets out of scope and is destroyed: the `ScopedBuffer` destructor is called. This causes the GenTL buffer contained in `buf` to be re-queued (given back) to the data stream of the grabber.

Example of program output

```
event timestamp: 77185931621 us, buffer timestamp: 77185919807 us
event timestamp: 77185951618 us, buffer timestamp: 77185939809 us
event timestamp: 77185971625 us, buffer timestamp: 77185959810 us
event timestamp: 77185991611 us, buffer timestamp: 77185979812 us
event timestamp: 77186011605 us, buffer timestamp: 77185999808 us
event timestamp: 77186031622 us, buffer timestamp: 77186019809 us
event timestamp: 77186051614 us, buffer timestamp: 77186039810 us
event timestamp: 77186071611 us, buffer timestamp: 77186059811 us
event timestamp: 77186091602 us, buffer timestamp: 77186079812 us
event timestamp: 77186111607 us, buffer timestamp: 77186099814 us
```

5.7. Relevant files

`include/EGrabber.h`	Main header. Includes all the other headers except `include/RGBConverter.h`. Defines `Euresys::EGrabber`, `Euresys::Buffer`, `Euresys::ScopedBuffer`
`include/EGrabberTypes.h`	Defines data types related to `Euresys::EGrabber`
`include/EGenTL.h`	Defines `Euresys::EGenTL`
`include/GenTL_v1_5.h`	Standard GenTL header. Defines standard types, functions and constants
`include/GenTL_v1_5_EuresysCustom.h`	Defines Coaxlink-specific constants
`include/RGBConverter.h`	Defines `Euresys::RGBConverter` helper class

6. Euresys GenApi scripts

The Euresys GenApi Script language is documented in a few GenApi scripts. For convenience, they are also included here.

6.1. doc/basics.js

```
// Euresys GenApi Script uses a syntax inspired by JavaScript, but is not
// exactly JavaScript. Using the extension .js for scripts is just a way to get
// proper syntax highlighting in text editors.
//
// This file describes the basics of Euresys GenApi Script. It can be executed
// by running 'gentl script <path-to-coaxlink-scripts-dir>/doc/basics.js', or
// more simply 'gentl script coaxlink://doc/basics.js'.


// Euresys GenApi Script is case-sensitive.

// Statements are always separated by semicolons (JavaScript is more
// permissive).

// Single-line comment

/* Multi-line comment
   (cannot be nested)
*/

// Function declaration
function multiply(a, b) {
    return a * b;
}

// Functions can be nested
function sumOfSquares(a, b) {
    function square(x) {
        return x * x;
    }
    return square(a) + square(b);
}

// Variable declaration
function Variables() {
    var x = 1;      // 1
    var y = 2 * x;  // 2
    var z;          // undefined
}

// Data types
function DataTypes() {
    // Primitive types: Boolean, Number, String, undefined
    function Booleans() {
        var x = true;
        var y = false;
    }
    function Numbers() {
        var x = 3.14159;
```

```
        var y = -1;
        var z = 6.022e23;
    }
    function Strings() {
        var empty = "";
        var x = "euresys";
        var y = 'coaxlink';
    }
    function Undefined() {
        // undefined is the type of variables without a value
        // undefined is also a special value
        var x; // undefined
        x = 1; // x has a value
        x = undefined; // x is now again undefined
    }
    // Objects: Object (unordered set of key/value pairs), Array (ordered list
    // of values), RegExp (regular expression)
    function Objects() {
        // Construction
        var empty = {};
        var x = { a: 1, b: 2, c: 3 };
        var y = { other: x };
        // Access to object properties
        var sum1 = x.a + x.b + x.c; // dot notation
        var sum2 = x['a'] + x['b'] + x['c']; // bracket notation
        // Adding properties
        x.d = 4;    // x: { a: 1, b: 2, c: 3, d: 4 }
        x["e"] = 5; // x: { a: 1, b: 2, c: 3, d: 4, e: 5 }
    }
    function Arrays() {
        // Construction
        var empty = [];
        var x = [3.14159, 2.71828];
        var mix = [1, false, "abc", {}];
        // Access to array elements
        var sum = x[0] + x[1]; // bracket notation
        // Adding elements
        x[2] = 1.61803; // x: [3.14159, 2.71828, 1.61803]
        x[4] = 1.41421; // x: [3.14159, 2.71828, 1.61803, undefined, 1.41421];
    }
    function RegularExpressions() {
        var x = /CXP[36]_X[124]/;
    }
}

// Like JavaScript, Euresys GenApi Script is a dynamically typed language. The
// type of a variable is defined by the value it holds, which can change.
function DynamicVariables() {
    var x = 1; // Number
    x = "x is now a string";
}

// Object types are accessed by reference.
function References() {
    var x = [3.14159, 2.71828]; // x is a reference to an array
    var y = x;                  // y is a reference to the same array
    assertEqual(x.length, y.length);
    assertEqual(x[0], y[0]);
    assertEqual(x[1], y[1]);
    y[2] = 1.61803;             // the array can be modified via any reference
    assertEqual(x[2], y[2]);

    function update(obj) {
        // objects (including arrays) are passed by reference
        obj.updated = true;
        obj.added = true;
    }
    var z = { initialized: true, updated: false };
    assertEqual(true, z.initialized);
    assertEqual(false, z.updated);
```

```
    assertEqual(undefined, z.added);
    update(z);
    assertEqual(true, z.initialized);
    assertEqual(true, z.updated);
    assertEqual(true, z.added);
}

// Supported operators
function Operators() {
    // From lowest to highest precedence:
    // Assignment operators: = += -= *= /=
    var x = 3;
    x += 2;
    x -= 1;
    x *= 3;
    x /= 5;
    assertEqual(2.4, x);
    // Logical OR: ||
    assertEqual(true, false || true);
    assertEqual('ok', false || 'ok');
    assertEqual('ok', 'ok' || 'ignored');
    // Logical AND: &&
    assertEqual(false, true && false);
    assertEqual(true, true && true);
    assertEqual('ok', true && 'ok');
    // Identity (strict equality) and non-identity (strict inequality): === !==
    assertEqual(true, 1 === 2 / 2);
    assertEqual(true, 1 !== 2);
    // Equality (==) and inequality (!=) JavaScript operators lead to confusing
    // and inconsistent conversions of their operands. They are not implemented
    // in Euresys GenApi Script.
    // Relational operators: < <= > >=
    assert(1 < 2);
    assert(1 <= 1);
    assert(2 > 1);
    assert(2 >= 2);
    // Addition and subtraction: + -
    assertEqual(1, 3 - 2);
    assertEqual(5, 2 + 3);
    assertEqual("abcdef", "abc" + "def"); // if one of the operands is of type
    assertEqual("abc123", "abc" + 123);   // string, all operands are converted
    assertEqual("123456", 123 + "456");   // to string, and concatenated
    // Multiplication and division: * /
    assertEqual(4.5, 3 * 3 / 2);
    // Prefix operators: ++ -- ! typeof
    var x = 0;
    assertEqual(1, ++x);
    assertEqual(1, x);
    assertEqual(0, --x);
    assertEqual(0, x);
    assertEqual(true, !false);
    assertEqual('boolean', typeof false);
    assertEqual('number', typeof 0);
    assertEqual('string', typeof '');
    assertEqual('undefined', typeof undefined);
    assertEqual('function', typeof function () {});
    assertEqual('object', typeof {});
    assertEqual('object', typeof []);
    assertEqual('object', typeof /re/);
    assertEqual('object', typeof null);
    // Postfix operators: ++ --
    var x = 0;
    assertEqual(0, x++);
    assertEqual(1, x);
    assertEqual(1, x--);
    assertEqual(0, x);
    // Function call: ()
    assertEqual(6, multiply(3, 2));
    // Member access: . []
    var obj = { a: 1 };
```

```
    assertEqual(1, obj.a);
    obj['4'] = 'four';
    assertEqual('four', obj[2*2]);
}

// Scope of variables
function OuterFunction() {
    var x = 'outer x';
    function Shadowing() {
        assertEqual(undefined, x);
        var x = 'inner x';
        assertEqual('inner x', x);
    }
    function Nested() {
        assertEqual('outer x', x);
        var y = 'not accessible outside Nested';
        x += ' changed in Nested';
    }
    function NoBlockScope() {
        var x = 1;
        assertEqual(1, x);
        if (true) {
            // The scope of variables is the function.
            // This variable x is the same as the one outside the if block.
            var x = 2;
        }
        assertEqual(2, x);
    }
    assertEqual('outer x', x);
    Shadowing();
    assertEqual('outer x', x);
    Nested();
    assertEqual('outer x changed in Nested', x);
    NoBlockScope();
}

// Loops
function Loops() {
    // for loops
    function ForLoops() {
        var i;
        var sum = 0;
        for (i = 0; i < 6; ++i) {
            sum += i;
        }
        assertEqual(15, sum);
    }
    // for..in loops: iterating over indices
    function ForInLoops() {
        var xs = [1, 10, 100, 1000];
        var sum = 0;
        for (var i in xs) {
            sum += xs[i];
        }
        assertEqual(1111, sum);
        var obj = { one: 1, two: 2 };
        var sum = 0;
        for (var p in obj) {
            sum += obj[p];
        }
        assertEqual(3, sum);
        var str = "Coaxlink";
        var sum = "";
        for (var i in str) {
            sum += str[i];
        }
        assertEqual("Coaxlink", sum);
    }
    // for..of loops: iterating over values
    function ForOfLoops() {
```

```
        var xs = [1, 10, 100, 1000];
        var sum = 0;
        for (var x of xs) {
            sum += x;
        }
        assertEqual(1111, sum);
        var obj = { one: 1, two: 2 };
        var sum = 0;
        for (var x of obj) {
            sum += x;
        }
        assertEqual(3, sum);
        var str = "Coaxlink";
        var sum = "";
        for (var c of str) {
            sum += c;
        }
        assertEqual("Coaxlink", sum);
    }
    function ContinueAndBreak() {
        var i;
        var sum = 0;
        for (i = 0; i < 100; ++i) {
            if (i === 3) {
                continue;
            } else if (i === 6) {
                break;
            } else {
                sum += i;
            }
        }
        assertEqual(0 + 1 + 2 + 4 + 5, sum);
    }
    ForLoops();
    ForInLoops();
    ForOfLoops();
    ContinueAndBreak();
}

function Exceptions() {
    var x;
    var caught;
    var finallyDone;
    function f(action) {
        x = 0;
        caught = undefined;
        finallyDone = false;
        try {
            x = 1;
            if (action === 'fail') {
                throw action;
            } else if (action === 'return') {
                return;
            }
            x = 2;
        } catch (e) {
            // Executed if a throw statement is executed.
            assertEqual(1, x);
            caught = e;
        } finally {
            // Executed regardless of whether or not a throw statement is
            // executed. Also executed if a return statement causes the
            // function to exit before the end of the try block.
            finallyDone = true;
        }
    }
    f('fail');
    assertEqual(1, x);
    assertEqual('fail', caught);
    assert(finallyDone);
```

```
    f('return');
    assertEqual(1, x);
    assert(!caught);
    assert(finallyDone);
    f();
    assertEqual(2, x);
    assert(!caught);
    assert(finallyDone);
}



// Run tests
References();
Operators();
OuterFunction();
Loops();
Exceptions();

function assertEqual(expected, actual) {
    if (expected !== actual) {
        throw 'expected: ' + expected + ', actual: ' + actual;
    }
}

function assert(condition) {
    if (!condition) {
        throw 'failed assertion';
    }
}
```

6.2. doc/builtins.js

```
// This file describes the builtins (functions or objects) of Euresys GenApi
// Script. It can be executed by running 'gentl script
// coaxlink://doc/builtins.js'.

// The builtin object 'console' contains a single function, log, which can be
// used to output text to the standard output.
console.log('Hello from ' + module.filename);
console.log('If several arguments are passed,', 'they are joined with spaces');

// The builtin object 'memento' contains the following functions: error,
// warning, notice, info, debug, verbose (each corresponding to a different
// verbosity level in Memento). They are similar to console.log, except that
// the text is sent to Memento.
memento.error('error description');
memento.warning('warning description');
memento.notice('important notification');
memento.info('message');
memento.debug('debug information');
memento.verbose('more debug information');

// Explicit type conversion/information functions:
console.log('Boolean(0) = ' + Boolean(0));               // false
console.log('Boolean(3) = ' + Boolean(3));               // true
console.log('Number(false) = ' + Number(false));         // 0
console.log('Number(true) = ' + Number(true));           // 1
console.log('Number("3.14") = ' + Number("3.14"));       // 3.14
console.log('Number("0x16") = ' + Number("0x16"));       // 22
console.log('Number("1e-9") = ' + Number("1e-9"));       // 1e-9
console.log('String(false) = ' + String(false));         // "false"
console.log('String(true) = ' + String(true));           // "true"
console.log('String(3.14) = ' + String(3.14));           // "3.14"
console.log('String([1, 2]) = ' + String([1, 2]));       // "1,2"
console.log('isNaN(0/0) = ' + isNaN(0/0));               // true
console.log('isNaN(Infinity) = ' + isNaN(Infinity));     // false
console.log('isRegExp(/re/) = ' + isRegExp(/re/));       // true
console.log('isRegExp("/re/") = ' + isRegExp("/re/"));   // false
console.log('Array.isArray({}) = ' + Array.isArray({})); // false
console.log('Array.isArray([]) = ' + Array.isArray([])); // true

// The builtin object 'Math' contains a few functions:
console.log('floor(3.14) = ' + Math.floor(3.14));
console.log('abs(-1.5) = ' + Math.abs(1.5));
console.log('pow(2, 5) = ' + Math.pow(2, 5));
console.log('log2(2048) = ' + Math.log2(2048));

// String manipulation
console.log('"Duo & Duo".replace(/Duo/,  "Quad") = "' +
            "Duo & Duo".replace(/Duo/,  "Quad") + '"'); // "Quad & Duo"
console.log('"Duo & Duo".replace(/Duo/g, "Quad") = "' +
            "Duo & Duo".replace(/Duo/g, "Quad") + '"'); // "Quad & Quad"
console.log('"Hello, Coaxlink".toLowerCase() = "' +
            "Hello, Coaxlink".toLowerCase() + '"');     // "hello, coaxlink"
console.log('"Coaxlink Quad G3".includes("Quad") = ' +
            "Coaxlink Quad G3".includes("Quad"));       // true
console.log('"Coaxlink Quad".includes("G3") = ' +
            "Coaxlink Quad".includes("G3"));            // false
console.log('"Coaxlink Quad G3".split(" ") = [' +
            "Coaxlink Quad G3".split(" ") + ']');       // [Coaxlink,Quad,G3]
console.log('"Coaxlink Quad G3".split("Quad") = [' +
            "Coaxlink Quad G3".split("Quad") + ']');    // [Coaxlink , G3]
console.log('["Mono", "Duo", "Quad"].join() = "' +
            ["Mono", "Duo", "Quad"].join() + '"');      // "Mono,Duo,Quad"
console.log('["Mono", "Duo", "Quad"].join(" & ") = "' +
            ["Mono", "Duo", "Quad"].join(" & ") + '"'); // "Mono & Duo & Quad"
```

```
// Utility functions
console.log('random(0,1): ' + random(0,1)); // random number between 0 and 1
sleep(0.5); // pause execution of script for 0.5 second

// The builtin function 'require' loads a script, executes it, and returns
// the value of the special 'module.exports' from that module.
var mod1 = require('./module1.js');
console.log('mod1.description: ' + mod1.description);
console.log('mod1.plus2(3): ' + mod1.plus2(3));
console.log('calling mod1.hello()...');
mod1.hello();

// 'require' can deal with:
// - absolute paths
//   var mod = require('C:\\absolute\\path\\some-module.js');
// - relative paths (paths relative to the current script)
//   var mod = require('./utils/helper.js');
// - coaxlink:// paths (paths relative to the directory where coaxlink scripts
//   are installed)
//   var mod = require(coaxlink://doc/builtins.js);
```

6.3. doc/grabbers.js

```
// This file describes the 'grabbers' object of Euresys GenApi Script. It can
// be executed by running 'gentl script coaxlink://doc/grabbers.js'.

// The builtin object 'grabbers' is a list of objects giving access to the
// available GenTL modules/ports.

// In most cases, 'grabbers' contains exactly one element. However, when using
// the 'gentl script' command, 'grabbers' contains the list of all devices.
// This makes it possible to configure several cameras and/or cards.

console.log("grabbers.length:", grabbers.length);

// Each item in 'grabbers' encapsulates all the ports related to one data
// stream:
//  TLPort          | GenTL producer
//  InterfacePort   | Coaxlink card
//  DevicePort      | local device
//  StreamPort      | data stream
//  RemotePort      | camera (if available)

var PortNames = ['TLPort', 'InterfacePort', 'DevicePort', 'StreamPort',
                 'RemotePort'];

// Ports are objects which provide the following textual information:
//  name            | one of PortNames
//  tag             | port handle type and value (as shown in memento traces)

for (var i in grabbers) {
    var g = grabbers[i];
    console.log('- grabbers[' + i + ']');
    for (var pn of PortNames) {
        var port = g[pn];
        console.log('  - ' + port.name + ' (' + port.tag + ')');
    }
}

// Ports also have the following functions to work on GenICam features:
//  get(f)          | get value of f
//  set(f,v)        | set value v to f
//  execute(f)      | execute f
//  features([re])  | get list of features [matching regular expression re]
//  $features([re]) | strict* variant of features([re])
//  ee(f,[re])      | get list of enum entries [matching regular expression re]
```

```
//                  | of enumeration f
//  $ee(f,[re])     | strict* variant of ee(f,[re])
//  has(f)          | test if f exists
//  has(f,v)        | test if f has an enum entry v
//  available(f)    | test if f is available
//  available(f,v)  | test if f has an enum entry v which is available
//  selectors(f)    | get list of features that act as selectors of f
//  attributes(...) | extract information from the XML file describing the port
//
// * by strict we mean that the returned list contains only nodes/values
//   that are available (as dictated by 'pIsAvailable' GenICam node elements)

if (grabbers.length) {
    var port = grabbers[0].InterfacePort;
    console.log('Playing with', port.tag);
    // get(f)
    console.log('- InterfaceID: ' + port.get('InterfaceID'));
    // set(f,v)
    port.set('LineSelector', 'TTLIO11');
    // execute(f)
    port.execute('DeviceUpdateList');
    // features(re)
    console.log('- Features matching \'PCIe\':');
    for (var f of port.features('PCIe')) {
        console.log('  - ' + f);
    }
    // $ee(f)
    console.log('- Available enum entries for LineSource:');
    for (var ee of port.$ee('LineSource')) {
        console.log('  - ' + ee);
    }
    for (var ix of [0, 1, 2, 3, 9]) {
        var ee = 'Device' + ix + 'Strobe';
        // has(f, v)
        if (port.has('LineSource', ee)) {
            console.log('- ' + ee + ' exists');
        } else {
            console.log('- ' + ee + ' does not exist');
        }
        // available(f, v)
        if (port.available('LineSource', ee)) {
            console.log('- ' + ee + ' is available');
        } else {
            console.log('- ' + ee + ' is not available');
        }
    }
    // selectors(f)
    console.log('- LineSource feature is selected by',
                port.selectors('LineSource'));
    // attributes()
    console.log('- attributes()');
    var attrs = port.attributes();
    for (var n in attrs) {
        console.log('  - ' + n + ': ' + attrs[n]);
    }
    // attributes(f)
    console.log('- attributes(\'LineFormat\')');
    var attrs = port.attributes('LineFormat');
    for (var n in attrs) {
        console.log('  - ' + n + ': ' + attrs[n]);
    }
    // attributes(f)
    var fmt = port.get('LineFormat');
    console.log('- attributes(\'LineFormat\', \'' + fmt + '\')');
    var attrs = port.attributes('LineFormat', fmt);
    for (var n in attrs) {
        console.log('  - ' + n + ': ' + attrs[n]);
    }
    // optional suffixes to integer or float feature names
    if (port.available('DividerToolSelector') &&
```

```
        port.available('DividerToolSelector', 'DIV1')) {
        var feature = 'DividerToolDivisionFactor[DIV1]';
        var suffixes = ['.Min', '.Max', '.Inc', '.Value'];
        console.log('- Accessing ' + suffixes + ' of ' + feature);
        for (var suffix of suffixes) {
            console.log( '  - ' + suffix + ': ' + port.get(feature + suffix));
        }
    }
}

// Camera ports (RemotePort) also have the following functions:
//  brRead(addr)    | read bootstrap register (32-bit big endian)
//  brWrite(addr,v) | write value to bootstrap register (32-bit big endian)

if (grabbers.length) {
    var port = grabbers[0].RemotePort;
    if (port) {
        console.log('Playing with', port.tag);
        var brStandard = 0x00000000;
        var brRevision = 0x00000004;
        var standard = port.brRead(brStandard);
        var revision = port.brRead(brRevision);
        if (0xc0a79ae5 === standard) {
            console.log('Bootstrap register "Standard" is OK (0xc0a79ae5)');
        } else {
            console.log('Bootstrap register "Standard" is ' + standard);
        }
        console.log('Bootstrap register "Revision" is ' + revision);
    }
}
```

6.4. doc/module1.js

```
// This file describes the special 'module' variable of Euresys GenApi Script.
// It can be executed by running 'gentl script coaxlink://doc/module1.js'. It
// is also dynamically loaded by the coaxlink://doc/builtins.js script.

// 'module' is a special per-module variable. It cannot be declared with var.
// It always exists, and contains a few items:
console.log('Started execution of "' + module.filename + '"');
console.log('This script is located in directory "' + module.curdir + '"');

// Modules can export values via module.exports (which is initialized as an
// empty object):
module.exports = { description: 'Example of Euresys GenApi Script module'
                 , plus2: function(x) {
                     return x + 2;
                   }
                 , hello: function() {
                     console.log('Hello from ' + module.filename);
                   }
                 };

console.log('module.exports contains: ');
for (var e in module.exports) {
    console.log('- ' + e + ' (' + typeof module.exports[e] + ')');
}

console.log('Completed execution of ' + module.filename);
```

7. EGrabber for MultiCam users

Concepts

MultiCam	EGrabber
Board	Interface
Channel	Device + Data stream
Surface	Buffer
Surface cluster (`MC_Cluster`)	Buffers announced to the data stream
-	Remote device (camera)
MultiCam parameters	GenApi set/get features
-	GenApi commands
CAM file	Euresys GenApi script
-	CallbackOnDemand
Callback functions	CallbackSingleThread
-	CallbackMultiThread

Initialization

```
// MultiCam
MCSTATUS status = McOpenDriver(NULL);
if (status != MC_OK) {
    ...
}
```

```
//EGrabber
Euresys::EGenTL gentl;
```

Channel creation

```
//MultiCam
MCSTATUS status;
MCHANDLE channel;

status = McCreate(MC_CHANNEL, &handle);
if (status != MC_OK) {
    ...
}
status = McSetParamInt(channel, MC_DriverIndex, CARD_INDEX);
if (status != MC_OK) {
    ...
```

```
}
status = McSetParamInt(channel, MC_Connector, CONNECTOR);
if (status != MC_OK) {
    ...
}
```

```
//EGrabber
Euresys::EGrabber<> grabber(gentl, CARD_INDEX, DEVICE_INDEX);
```

Surface creation (automatic)

```
//MultiCam
status = McSetParamInt(channel, MC_SurfaceCount, BUFFER_COUNT);
if (status != MC_OK) {
    ...
}
```

```
//EGrabber
grabber.reallocBuffers(BUFFER_COUNT);
```

Surface creation (manual)

```
//MultiCam
for (size_t i = 0; i < BUFFER_COUNT; ++i) {
    MCHANDLE surface;
    MCSTATUS status;
    void *mem = malloc(BUFFER_SIZE);
    if (!mem) {
        ...
    }
    status = McCreate(MC_DEFAULT_SURFACE_HANDLE, &surface);
    if (status != MC_OK) {
        ...
    }
    status = McSetParamInt(surface, MC_SurfaceSize, BUFFER_SIZE);
    if (status != MC_OK) {
        ...
    }
    status = McSetParamPtr(surface, MC_SurfaceAddr, mem);
    if (status != MC_OK) {
        ...
    }
    status = McSetParamPtr(surface, MC_SurfaceContext, USER_PTR[i]);
    if (status != MC_OK) {
        ...
    }
    status = McSetParamInst(channel, MC_Cluster + i, surface);
    if (status != MC_OK) {
        ...
    }
}
```

```
//EGrabber
for (size_t i = 0; i < BUFFER_COUNT; ++i) {
    void *mem = malloc(BUFFER_SIZE);
    if (!mem) {
        ...
    }
    grabber.announceAndQueue(Euresys::UserMemory(mem, BUFFER_SIZE, USER_PTR[i]));
}
```

Surface cluster reset

```
//MultiCam
MCSTATUS status;
for (size_t i = 0; i < BUFFER_COUNT; ++i) {
    MCHANDLE surface;
    status = McGetParamInst(channel, MC_Cluster + i, &surface);
    if (status != MC_OK) {
        ...
    }
    status = McSetParamInt(surface, MC_SurfaceState, MC_SurfaceState_FREE);
    if (status != MC_OK) {
        ...
    }
}
status = McSetParamInt(channel, MC_SurfaceIndex, 0);
if (status != MC_OK) {
    ...
}
```

```
//EGrabber
grabber.resetBufferQueue();
```

Frame grabber configuration

MultiCam	EGrabber
`McSetParamStr(H, MC_CamFile, filepath)`	`grabber.runScript(filepath)`
-	`grabber.runScript(script)`
`McSetParamInt(H, id, value)` or `McSetParamNmInt(H, name, value)`	`grabber.setInteger<M>(name, value)`
`McSetParamFloat(H, id, value)` or `McSetParamNmFloat(H, name, value)`	`grabber.setFloat<M>(name, value)`
`McSetParamStr(H, id, value)` or `McSetParamNmStr(H, name, value)`	`grabber.setString<M>(name, value)`

where `H` is a `MC_HANDLE` (the global `MC_CONFIGURATION` handle, a board handle, or a channel handle), and `M` specifies the target module (either `SystemModule`, `InterfaceModule`, `DeviceModule`, or `StreamModule`).

Camera configuration

MultiCam	EGrabber
-	`grabber.runScript(filepath)`
-	`grabber.runScript(script)`
-	`grabber.setInteger<RemoteModule>(name, value)`, `grabber.setFloat<RemoteModule>`

MultiCam	EGrabber
	(name, value), or grabber.setString<RemoteModule> (name, value)

Script files

```
//MultiCam
; CAM file
ChannelParam1 = Value1;
ChannelParam2 = Value2;
```

```
//EGrabber
// Euresys GenApi Script
var grabber = grabbers[0];
grabber.DevicePort.set('DeviceFeature1', Value1);
grabber.DevicePort.set('DeviceFeature2', Value2);
grabber.RemotePort.set('CameraFeatureA', ValueA);
```

Acquisition start/stop

```
//MultiCam
// start "live"
McSetParamInt(channel, MC_GrabCount, MC_INFINITE);
McSetParamInt(channel, MC_ChannelState, MC_ChannelState_ACTIVE);
// stop
McSetParamInt(channel, MC_ChannelState, MC_ChannelState_IDLE);
// grab 10 images
McSetParamInt(channel, MC_GrabCount, 10);
McSetParamInt(channel, MC_ChannelState, MC_ChannelState_ACTIVE);
```

```
//EGrabber
// start "live"
grabber.start();
// stop
grabber.stop();
// grab 10 images
grabber.start(10);
```

Synchronous (blocking) buffer reception

```
//MultiCam
MCSTATUS status;
MCSIGNALINFO info;
// wait for a surface
status = McWaitSignal(channel, MC_SIG_SURFACE_PROCESSING, timeout, &info);
if (status != MC_OK) {
    ...
}
MCHANDLE surface = info.SignalInfo;
// process surface
...
// make surface available for new images
status = McSetParamInt(surface, MC_SurfaceState, MC_SurfaceState_FREE);
if (status != MC_OK) {
    ...
}
```

```
//EGrabber
// wait for a buffer
```

```
Buffer buffer = grabber.pop(timeout);
// process buffer
...
// make buffer available for new images
buffer.push(grabber);
```

```
//EGrabber
{
    // wait for a buffer
    ScopedBuffer buffer(grabber, timeout);
    // process buffer
    ...
    // ScopedBuffer destructor takes care of making buffer available for new images
}
```

Callbacks

```
//MultiCam
class MyChannel {
    public:
        MyChannel() {
            // create and configure channel
            ...
            // enable "SURFACE_PROCESSING" events
            status = McSetParamInt(channel, MC_SignalEnable + MC_SIG_SURFACE_PROCESSING,
                                           MC_SignalEnable_ON);
            if (status != MC_OK) {
                ...
            }
            // enable "END_EXPOSURE" events
            status = McSetParamInt(channel, MC_SignalEnable + MC_SIG_END_EXPOSURE,
                                           MC_SignalEnable_ON);
            if (status != MC_OK) {
                ...
            }
            // register "extern C" callback function
            MCSTATUS status = McRegisterCallback(channel, GlobalCallbackFunction, this);
            if (status != MC_OK) {
                ...
            }
        }

        void onEvent(MCSIGNALINFO *info) {
            switch (info->Signal) {
                case MC_SIG_SURFACE_PROCESSING:
                    MCHANDLE surface = info.SignalInfo;
                    // process surface
                    ...
                    break;
                case MC_SIG_END_EXPOSURE:
                    // handle "END_EXPOSURE" event
                    ...
                    break;
            }
        }

    private:
        MCHANDLE channel;
};

void MCAPI GlobalCallbackFunction(MCSIGNALINFO *info) {
    if (info && info->Context) {
        MyGrabber *grabber = (MyGrabber *)info->Context;
        grabber->onEvent(info);
    }
};
```

```
//EGrabber
class MyGrabber : public EGrabber<CallbackSingleThread> {
    public:
        MyGrabber(EGenTL &gentl) : EGrabber<CallbackSingleThread>(gentl) {
            // configure grabber
            ...
            // enable "NewBuffer" events
            enableEvent<NewBufferData>();
            // enable "Cic" events
            enableEvent<CicData>();
        }

    private:
        virtual void onNewBufferEvent(const NewBufferData& data) {
            ScopedBuffer buffer(*this, data);
            // process buffer
            ...
        }
        virtual void onCicEvent(const CicData &data) {
            // handle "Cic" event
            ...
        }
};
```

Synchronous (blocking) event handling

```
//MultiCam
class MyChannel {
    public:
        MyChannel() {
            // create and configure channel
            ...
            // enable "END_EXPOSURE" events
            status = McSetParamInt(channel, MC_SignalEnable + MC_SIG_END_EXPOSURE,
                                           MC_SignalEnable_ON);
            if (status != MC_OK) {
                ...
            }
        }

        void waitForEvent(uint32_t timeout) {
            // wait for an event
            MCSTATUS status = McWaitSignal(channel, MC_SIG_END_EXPOSURE, timeout, &info);
            if (status != MC_OK) {
                ...
            }
            // handle "END_EXPOSURE" event
            ...
        }

    private:
        ...
};
```

```
//EGrabber
class MyGrabber : public EGrabber<CallbackOnDemand> {
    public:
        MyGrabber(EGenTL &gentl) : EGrabber<CallbackOnDemand>(gentl) {
            // configure grabber
            ...
            // enable "Cic" events
            enableEvent<CicData>();
        }

        void waitForEvent(uint64_t timeout) {
            // wait for an event
            processEvent<CicData>(timeout);
```

```
        }

    private:
        // onCicEvent is called by processEvent when a "Cic" event occurs
        virtual void onCicEvent(const CicData &data) {
            // handle "Cic" event
            ...
        }
};
```

8. .NET assembly

`EGrabber` can be used in .NET languages (C#, VB.NET, etc.) via a .NET assembly named `Coaxlink_NetApi.dll`

8.1. A first example

This example creates a grabber and displays basic information about the interface, device, and remote device modules it contains. This is the C# version of the first C++ `EGrabber` example:

```
using System;
namespace FirstExample {
    class ShowInfo {
        const int CARD_IX = 0;
        const int DEVICE_IX = 0;

        static void showInfo() {
            using (Euresys.EGenTL gentl = new Euresys.EGenTL()) {                          // 1
                using (Euresys.EGrabberCallbackOnDemand grabber =
                       new Euresys.EGrabberCallbackOnDemand(gentl, CARD_IX, DEVICE_IX)) {  // 2
                    String card = grabber.getStringInterfaceModule("InterfaceID");         // 3
                    String dev = grabber.getStringDeviceModule("DeviceID");                // 4
                    long width = grabber.getIntegerRemoteModule("Width");                  // 5
                    long height = grabber.getIntegerRemoteModule("Height");                // 5

                    System.Console.WriteLine("Interface:    {0}", card);
                    System.Console.WriteLine("Device:       {0}", dev);
                    System.Console.WriteLine("Resolution:   {0}x{1}", width, height);
                }
            }
        }

        static void Main() {
            try {                                                                          // 6
                showInfo();
            } catch (System.Exception e) {                                                 // 6
                System.Console.WriteLine("error: {0}", e.Message);
            }
        }
    }
}
```

1. Create a `Euresys.EGenTL` object. This involves the following operations:
   - locate and dynamically load the Coaxlink GenTL producer (`coaxlink.cti`);
   - retrieve pointers to the functions exported by `coaxlink.cti`;
   - initialize `coaxlink.cti`.
2. Create a `Euresys.EGrabberCallbackOnDemand` object. The constructor needs the `gentl` object we've just created. It also takes as optional arguments the indices of the interface and device to use.

3. Use "GenApi" on page 6 to find out the ID of the Coaxlink card. Notice the `InterfaceModule` suffix in `getString`**`InterfaceModule`**. This indicates that we want an answer from the interface module.
4. Similarly, find out the ID of the device. This time, we use `getString`**`DeviceModule`** to target the device module.
5. Finally, read the camera resolution. This time, we use `getInteger`**`RemoteModule`** because values must be read from the camera.
6. `EGrabber` uses exceptions to report errors, so we wrap our code inside a `try ... catch` block.

Example of program output

```
Interface:    PC1633 - Coaxlink Quad G3 (2-camera) - KQG00014
Device:       Device0
Resolution:   4096x4096
```

8.2. Differences between C++ and .NET EGrabber

Terms in *`ITALIC`* are placeholders:

- *`MODULE`* can be replaced by `InterfaceModule`, `DeviceModule`...
- *`EVENT_DATA`* can be replaced by `NewBufferData`, `CicData`...

EGrabber classes

C++	.NET
`EGrabber<>`	-
`EGrabber<CallbackOnDemand>`	`EGrabberCallbackOnDemand`
`EGrabber<CallbackSingleThread>`	`EGrabberCallbackSingleThread`
`EGrabber<CallbackMultiThread>`	`EGrabberCallbackMultiThread`

EGrabber Methods

C++	.NET
`getInfo<`*`MODULE`*`,`*`TYPE`*`>(cmd)`	`getInfo`*`MODULE`*`(cmd, out ...)`
`getInteger<`*`MODULE`*`>(f)`	`getInteger`*`MODULE`*`(f)`
`getFloat<`*`MODULE`*`>(f)`	`getFloat`*`MODULE`*`(f)`

C++	.NET
`getString<MODULE>(f)`	`getStringMODULE(f)`
`getStringList<MODULE>(f)`	`getStringListMODULE(f)`
`setInteger<MODULE>(f, v)`	`setIntegerMODULE(f, v)`
`setFloat<MODULE>(f, v)`	`setFloatMODULE(f, v)`
`setString<MODULE>(f, v)`	`setStringMODULE(f, v)`
`execute<MODULE>(f)`	`executeMODULE(f)`
`enableEvent<EVENT_DATA>()`	`enableEVENT_DATAEvent(f)`
`disableEvent<EVENT_DATA>()`	`disableEVENT_DATAEvent(f)`

Callbacks

In .NET, callbacks are defined as delegates:

```
grabber.onNewBufferEvent = delegate ...
grabber.onDataStreamEvent = delegate ...
grabber.onCicEvent = delegate ...
grabber.onIoToolboxEvent = delegate ...
grabber.onCxpInterfaceEvent = delegate ...
```

A complete example is given in the next section.

8.3. Single thread callbacks

This program displays basic information about CIC events generated by a grabber, using the `CallbackSingleThread` model. This is the C# version of the C++ `CallbackSingleThread` example:

```
using System;

namespace Callbacks {
    class CallbackExample {
        static void showEvents(Euresys.EGrabberCallbackSingleThread grabber) {
            grabber.runScript("config.js");                                          // 1

            grabber.onCicEvent = delegate(Euresys.EGrabberCallbackSingleThread g,    // 2
                                          Euresys.CicData data) {
                System.Console.WriteLine("timestamp: {0} us, {1}",                   // 3
                                         data.timestamp, data.numid);
            };                                                                       // 4

            grabber.enableCicDataEvent();                                            // 5

            grabber.reallocBuffers(3);                                               // 6
            grabber.start();                                                         // 6
            while (true) {                                                           // 6
            }
        }

        static void Main() {
            try {
                using (Euresys.EGenTL gentl = new Euresys.EGenTL()) {
                    using (Euresys.EGrabberCallbackSingleThread grabber =
```

```
                        new Euresys.EGrabberCallbackSingleThread(gentl)) {
                    showEvents(grabber);
                }
            }
        } catch (System.Exception e) {
            System.Console.WriteLine("error: {0}", e.Message);
        }
    }
}
```

1. Run a `config.js` script which should:
   - properly configure the camera and frame grabber;
   - enable notifications for CIC events.
2. Register the callback function for *CIC* events:
   - create a `delegate` that will be called by `EGrabber` when a *CIC* event occurs; this delegate will be called with two arguments: the grabber and the `CicData` containing information about the event;
   - set the grabber's `onDataStreamEvent` to this `delegate` function.
3. In the body of the callback function, simply display basic information about the event.
4. This ends the definition of the `onCicEvent` callback function.
5. Enable `onCicEvent` callbacks.
6. Start the grabber and enter an infinite loop. *CIC* events will be notified in a dedicated thread.

Example of program output

```
timestamp: 2790824897 us, EVENT_DATA_NUMID_CIC_CAMERA_TRIGGER_RISING_EDGE
timestamp: 2790824897 us, EVENT_DATA_NUMID_CIC_STROBE_RISING_EDGE
timestamp: 2790824902 us, EVENT_DATA_NUMID_CIC_CXP_TRIGGER_ACK
timestamp: 2790825897 us, EVENT_DATA_NUMID_CIC_STROBE_FALLING_EDGE
timestamp: 2790830397 us, EVENT_DATA_NUMID_CIC_CAMERA_TRIGGER_FALLING_EDGE
timestamp: 2790830401 us, EVENT_DATA_NUMID_CIC_CXP_TRIGGER_ACK
timestamp: 2790842190 us, EVENT_DATA_NUMID_CIC_ALLOW_NEXT_CYCLE
timestamp: 2790842190 us, EVENT_DATA_NUMID_CIC_CAMERA_TRIGGER_RISING_EDGE
timestamp: 2790842191 us, EVENT_DATA_NUMID_CIC_STROBE_RISING_EDGE
timestamp: 2790842195 us, EVENT_DATA_NUMID_CIC_CXP_TRIGGER_ACK
```

9. Definitions

Acronyms

CIC

Camera and Illumination Controller

CTI

Common Transport Interface

CXP

CoaXPress

EMVA

European Machine Vision Association

PFNC

Pixel Format Naming Convention

SFNC

Standard Features Naming Convention

Glossary

Buffer module

GenTL module that represents a memory buffer. Buffers must be announced to the data stream that will fill them with image data.

Callback model

Defines when and where (in which thread) callback functions are executed.

One of `CallbackOnDemand`, `CallbackSingleThread`, `CallbackMultiThread`.

Camera and illumination controller

Part of Coaxlink card that controls a camera and its associated illumination devices.

Hosted in the device module.

CoaXPress

High speed digital interface standard that allows the transmission of data from a device (e.g., a camera) to a host (e.g., a frame grabber inside a computer) over one or several coaxial cables.

Common transport interface

GenTL producer.

Data stream module

GenTL module that handles buffers.

Device module

GenTL module that contains the frame grabber settings relating to the camera.

Parent of the data stream module.

Sibling of the *remote device*.

GenApi

The GenICam standard that deals with camera and frame grabber configuration.

GenApi feature

Camera or frame grabber feature, defined in a register description.

Either a *set/get* parameter, or a command with side effects.

GenICam

Set of EMVA standards. It consists of GenApi, GenTL, the SFNC and the PFNC.

GenTL

The GenICam standard that deals with data transport. TL stands for Transport Layer.

GenTL producer

Software library that implements the GenTL API.

File with the `cti` extension (e.g., `coaxlink.cti`).

Info command

Numerical identifier used to query a specific piece of information from a GenTL module. Info commands are defined either in the standard GenTL header file, or in a vendor-specific header file (e.g., info commands specific to Coaxlink are defined in `include/GenTL_v1_5_EuresysCustom.h`).

Interface module

GenTL module that represents a frame grabber.

Parent of the device module.

I/O toolbox

Part of Coaxlink card that controls digital I/O lines and implements tools such as rate converters, delay lines, etc.

Hosted in the interface module.

Register description

XML file mapping low-level hardware registers to camera or frame grabber features.

Remote device

Camera connected to a frame grabber.

The term *remote* is used to distinguish this from the GenTL device module.

System

GenTL module that represents the GenTL producer.

Also known as TLSystem.

Parent of the interface module.

Timestamp

The time at which an event occurs.

For Coaxlink, timestamps are always 64-bit integers and are expressed as the number of microseconds that have elapsed since the computer was started.