GenICam GenTL Standard

Version 1.0

Document Version 1.0.190

Contents

Figures

Tables

Changes

Version	Date	Author	Description
0.1	May 1st 2007	Rupert Stelz, STEMMER IMAGING	1st Version
0.2	July 18th 2007	Rupert Stelz, STEMMER IMAGING	Added Enums Added Std Features Added AcqMode Drawings
0.3	November 2007	Sub Committee: Rupert Stelz, STEMMER IMAGING Sascha Dorenbeck, STEMMER IMAGING Jan Becvar, Leutron Vision Carsten Bienek, IDS Francois Gobeil, Pleora Technologies Christoph Zierl, MVTec	Applied changes as discussed on the last meeting in Ottawa
0.4	Januar 2008	Sub Committee	Removed EventGetDataEx and CustomEvent functionality Added comments from IDS, Matrix Vision, Matrox, Pleora, Leutron Vision, STEMMER IMAGING

1 Introduction

1.1 Purpose

The goal of the GenICam GenTL standard is to provide a generic way to enumerate devices known to a system, communicate with one or more devices and, if possible, stream data from the device to the host independent from the underlying transport technology. This allows a third party software to use different technologies to control cameras and to acquire data in a transport layer agnostic way.

The core of the GenICam GenTL standard is the definition of a generic Transport Layer Interface (TLI). This software interface between the transport technology and a third party software is defined by a C interface together with a defined behavior and a set of standardized feature names and their meaning. To access these features the GenICam GenApi module is used.

The GenICam GenApi module defines an XML description file format to describe how to access and control device features. The Standard Feature Naming Convention defines the behavior of these features.

The GenTL software interface does not cover any device-specific functionality of the remote device except the one to establish communication. The GenTL provides a port to allow access to the remote device features via the GenApi module.

This makes the GenTL the generic software interface to communicate with devices and stream data from them. The combination of GenApi and GenTL provides a complete software architecture to access devices, for example cameras.

1.2 Committee

The following members of the GenICam Standard Group are members of the GenTL subcommittee that is responsible for developing the GenICam GenTL Standard:

- IDS
- Leutron Vision
- MATRIX VISION
- Matrox
- MVTec Software
- Pleora Technologies
- STEMMER IMAGING

1.3 Definitions and Acronyms

1.3.1 Definitions

Term	Description
GenApi	GenICam module defining the GenApi XML Schema
GenTL	Generic Transport Layer Interface
GenTL Consumer	A library or application using an implementation of a Transport Layer Interface
GenTL Producer	Transport Layer Interface implementation
Signaling	Mechanism to notify the calling GenTL Consumer of an asynchronous event.
Configuration	Configuration of a module through the GenTL Port functions, a GenApi compliant XML description and the GenTL Standard Feature Naming Convention.

1.3.2 Acronyms

Term	Description
GenICam	Generic Interface to Cameras
GenTL	Generic Transport Layer
GigE	Gigabit Ethernet
PC	Personal Computer
TLI	Generic Transport Layer Interface
CTI	Common Transport Interface
CL	Camera Link
IIDC	1394 Trade Association Instrumentation and Industrial Control Working Group, Digital Camera Sub Working Group.
USB	Universal Serial Bus
UVC	USB Video Class

1.4 References

GenICam Standard Version 1.0 www.genicam.org

ISO C Standard (ISO/IEC 9899:1990(E))

GigE Vision Standard

2 Architecture

This section provides a high level view of the different components of the GenICam GenTL standard.

2.1 Overview

The goal of GenTL is to provide an agnostic transport layer interface to acquire image or other data and to communicate with a device. It is not its purpose to configure the device except for the transport related features – even if it must be indirectly used in order to communicate configuration information to and from the device.

2.1.1 GenICam GenTL

The standard text's primary concern is the definition of the GenTL Interface and its behavior. However, it is also important to understand the role of the GenTL in the whole GenICam system.

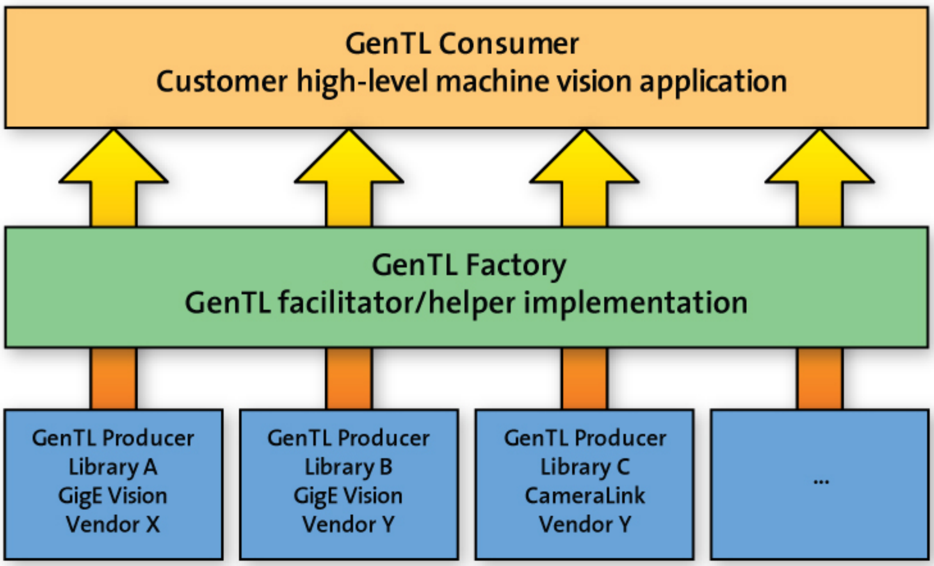


Figure 2-1: GenTL Consumer and GenTL Producer

When used alone, GenTL is used to identify two different entities: the GenTL Producer and the GenTL Consumer.

A GenTL Producer is a software driver with the extension implementing the GenTL Interface to enable an application or a software library to access and configure hardware in a generic way and to stream image data from a device.

A GenTL Consumer is any software which can use one or multiple GenTL Producers via the defined GenTL Interface. This can be for example an application or a software library.

2.1.2 GenICam GenApi

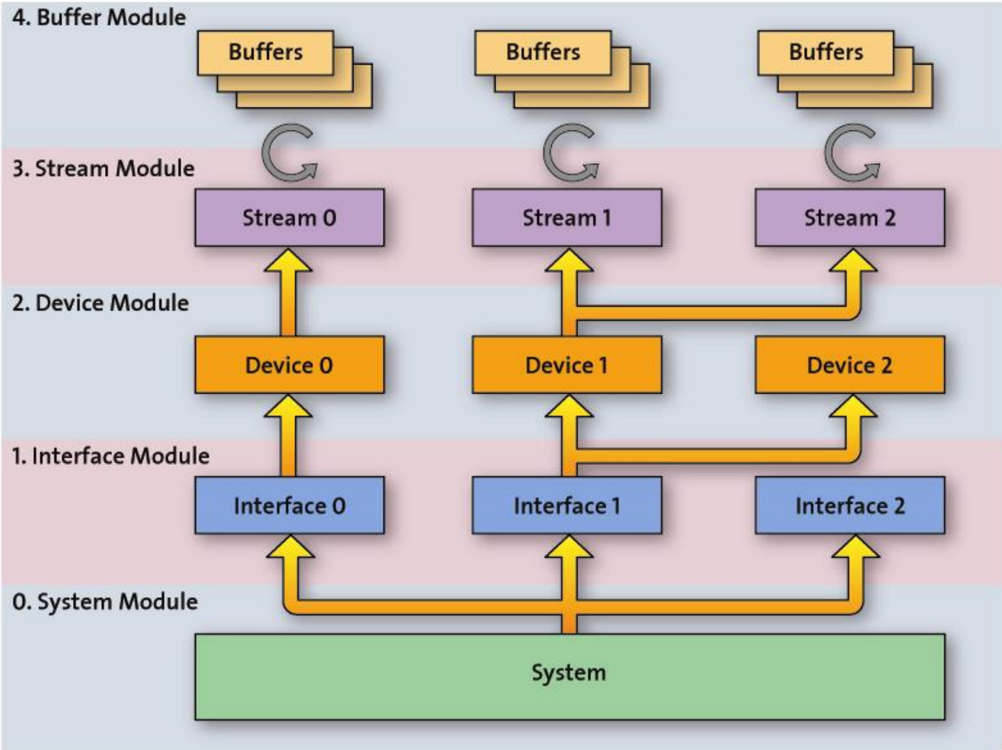
It is strongly recommended not to use the GenApi module inside the GenTL Producer implementations. If it is used internally no access to it may be given through the C interface. Some reasons are:

- **Retrieval of the correct GenICam XML file**: for the device configuration XML there is no unique way a GenTL Producer can create a node map that will be always identical to the one used by the application. Even if in most cases the XML is retrieved from the device, it cannot be assumed that it will always be the case.
- **GenICam XML description implementation**: there is no standardized implementation. GenApi is only a reference implementation, not a mandatory standard. User implementations in the same or in a different language may be used to interpret GenApi XML files. Even if the same implementation is used, the GenTL Producer and Consumer may not even use the same version of the implementation.
- **Caching**: when using another instance of an XML description inside the GenTL Producer, unwanted cache behavior may occur because both instances will be maintaining their own local, disconnected caches.

2.2 GenTL Modules

The GenTL standard defines a layered structure for libraries implementing the GenTL Interface. Each layer is defined in a module. The modules are presented in a tree structure with the System module as its root.

Figure 2-2: GenTL Module hierarchy



2.2.1 System Module

For every GenTL Consumer the System module as the root of the hierarchy is the entry point to a GenTL Producer software driver. It represents the whole system (not global, just the whole system of the GenTL Producer driver) on the host side from the GenTL libraries point of view.

The main task of the System module is to enumerate and instantiate available interfaces covered by the implementation.

The System module also provides signaling capability and configuration of the module's internal functionality to the GenTL Consumer.

It is possible to have a single GenTL Producer incorporating multiple transport layer technologies and to express them as different Interface modules. In this case the transport layer technology of the System module must be 'Mixed' and the child Interface modules expose their actual transport layer technology. In this case the first interface could then be a Camera Link frame grabber board and the second interface an IIDC 1394 controller.

2.2.2 Interface Module

An Interface module represents one physical interface in the system. For Ethernet based transport layer technologies this would be a Network Interface Card (NIC); for a Camera Link based implementation this would be one frame grabber board. The enumeration and instantiation of available devices on this interface is the main role of this module. The Interface module also presents Signaling and module configuration capabilities to the GenTL Consumer.

One system may contain zero, one or multiple interfaces. An interface is always only of one transport layer technology. It is not allowed to have e.g. a GEV camera and a Camera Link camera on one interface. There is no logical limitation on the number of interfaces addressed by the system. This is limited solely by the hardware used.

2.2.3 Device Module

The Device module represents the GenTL Producers' proxy for one physical remote device. The responsibility of the Device module is to enable the communication with the remote device and to enumerate and instantiate Data Stream modules. The Device module also presents Signaling and module configuration capabilities to the GenTL Consumer.

One Interface module can contain zero, one or multiple Device module instances. A device is always of one transport layer technology. There is no logical limitation on the number of devices attached to an interface. This is limited solely by the hardware used.

2.2.4 Data Stream Module

A single (image) data stream from a remote device is represented by the Data Stream module. The purpose of this module is to provide the acquisition engine and to maintain the internal buffer pool. Beside that the Data Stream module also presents Signaling and module configuration capabilities to the GenTL Consumer.

One device can contain zero, one or multiple data streams. There is no logical limitation on the number of streams a device can have. This is limited solely by the hardware used and the implementation.

2.2.5 Buffer Module

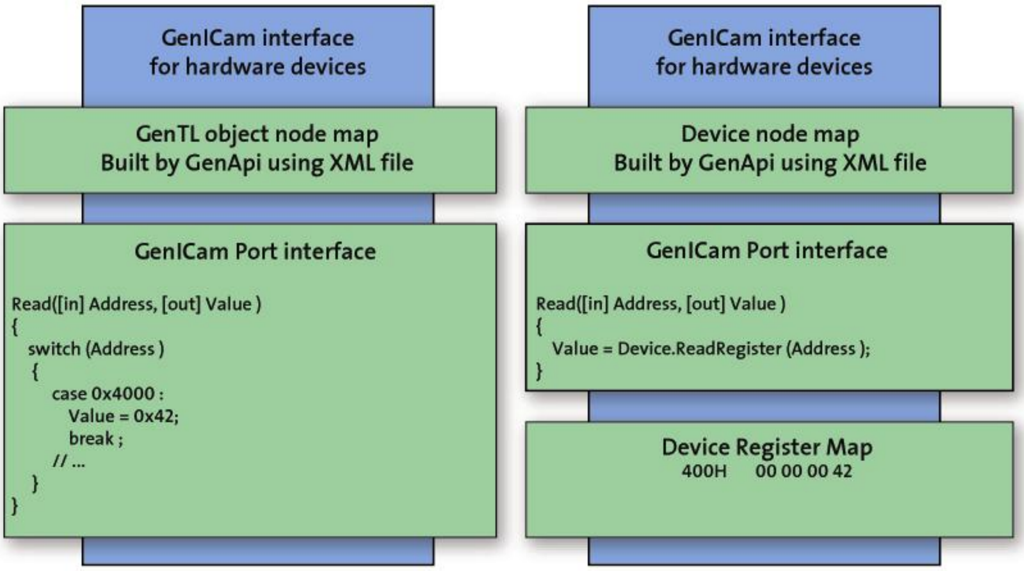
The Buffer module encapsulates a single memory buffer. Its purpose is to act as the target for acquisition. The memory of a buffer can be user allocated or GenTL Producer allocated. The latter could be pre-allocated system memory. The Buffer module also presents Signaling and module configuration capabilities to the GenTL Consumer.

To enable streaming of data at least one buffer has to be announced to the Data Stream module instance and placed into the input buffer pool.

2.3 GenTL Module Common Parts

Access and compatibility between GenTL Consumers and GenTL Producers is ensured by the C interface and the description of the behavior of the modules, the Signaling, the Configuration and the acquisition engine.

Figure 2-3: GenICam GenTL interface (C and GenApi Feature-interface)



The GenTL Producer driver consists of three logical parts: the C interface, the Configuration interface and the Events interface (signaling). The interfaces are detailed as follows:

2.3.1 C interface

The C interface provides the entry point of the GenTL Producer. It enumerates and creates all module instances. It includes the acquisition handled by the Data Stream module. The Signaling and Configuration interfaces of the module are also accessed by GenTL Consumer through the C interface. Thus it is possible to stream an image by just using the C interface independent of the underlying technology. This also means that the default state of a GenTL Provider should ensure the ability to open a device and receive data from it.

A C interface was chosen because of multiple reasons:

- **Support of multiple client languages:** a C interface library can be imported by many programming languages. Basic types can be marshaled easily between the languages and modules (different heaps, implementation details).
- **Dynamic loading of libraries:** it is easily possible to dynamically load and call C style functions. This enables the implementation of a GenTL Consumer dynamically loading one or more GenTL Producers at runtime.
- **Upgradeability:** a C library can be designed in a way that it is binary compatible to earlier versions. Thus the GenTL Consumer does not need to be recompiled if a version change occurs.

Although a C interface was chosen because of the reasons mentioned above, the actual GenTL Producer implementation can be done in an object-oriented language. Except for the global functions, all interface functions work on handles which can be mapped to objects.

Any programming language which can export a library with a C interface can be used to implement a GenTL Producer.

To guarantee interchangeability of GenTL Producers and GenTL Consumers no language specific feature except the ones compatible to ANSII C may be used in the interface of the GenTL Producer.

2.3.2 Configuration

Each module provides GenTL Port functionality so that the GenICam GenApi (or any other similar, non-reference implementations) can be used to access a module's configuration. The basic operations on a GenTL Producer implementation can be done with the C interface without using specific module configuration. More complex or implementation-specific access can be done via the flexible GenApi Feature interface using the GenTL Port functionality and the provided GenApi XML description.

Each module brings this XML description along with which the module's port can be used to read and/or modify settings in the module. To do that each module has its own virtual register map which can be accessed by the Port functions. Thus the generic way of accessing the configuration of a remote device has been extended to the transport layer modules themselves.

2.3.3 Signaling (Events)

Each module provides the possibility to notify the GenTL Consumer of certain events. As an example, a "NewBuffer" event can be raised/signaled if new image data has arrived from a remote device. The number of events supported for a specific module depends on the module and its implementation.

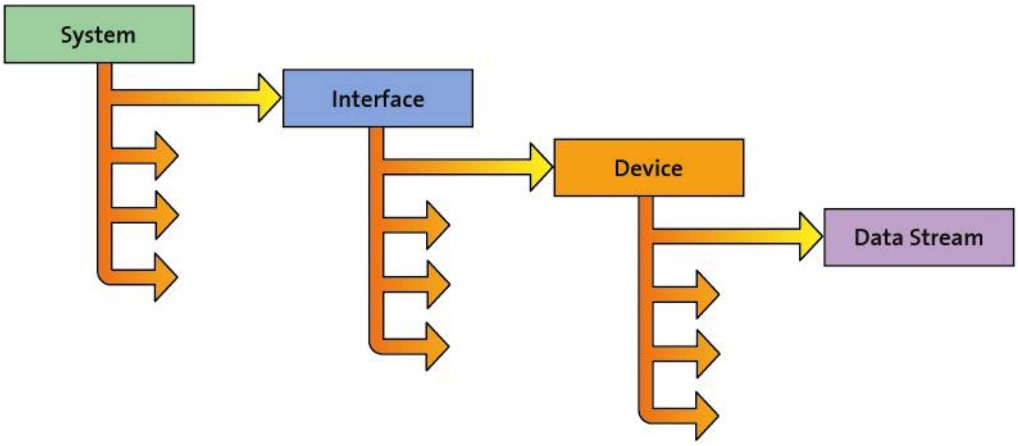
The C interface enables the GenTL Consumer to register events on a module. The event object used is platform and implementation dependent.

3 Module Enumeration and Instantiation

The behavior described below is seen from a single process' point of view. A GenTL Producer implementation must make sure that every process that is allowed to access the resources has this separated view on the hardware without the need to know that other processes are involved.

For a detailed description of the C functions and data types see chapter 6 Software Interface page 1ff. For how to configure a certain module or get notified on events see chapter 4 Configuration and Signaling page 29ff.

Figure 3-1: Enumeration hierarchy of a GenTL Producer



3.1 Setup

Before the System module can be opened and any operation can be performed on the GenTL Producer driver the `GCInitLib` function must be called. This must be done at least once per process. After the System module has been closed (when e.g. the GenTL Consumer is closed) the `GCCloseLib` function must be called to properly free all resources. If the library is used after `GCCloseLib` was called the `GCInitLib` must be called again.

3.2 System

The System module is always the entry point for the calling GenTL Consumer to the GenTL Producer. With the functions present here all available hardware interfaces in the form of an Interface module can be enumerated.

By calling the `TLOpen` function the `TL_HANDLE` to work on the System module's functions can be retrieved. The `TL_HANDLE` obtained from a successful call to the `TLOpen` function will be needed for all successive calls to other functions belonging to the System module.

Before doing that, the `GCGetInfo` function might be called to retrieve the basic information about the GenTL Producer implementation without opening the system module.

Each GenTL Producer driver exposes only a single System instance in an operating system process space. If a System module is requested more than once from within the same process space an error `GC_ERR_RESOURCE_IN_USE` is returned. If a GenTL Producer allows access from multiple processes it has to take care of the inter-process-communication and must handle the book-keeping of instantiated system modules. If it does not allow this kind of access it must return an appropriate error code whenever an attempt to create a second System module instance from another operating system process is made.

The System module does no reference counting within a single process. Thus even when a System module handle is requested twice from within a single process space, the second call will return an error `GC_ERR_RESOURCE_IN_USE`. The first call to the close function from within that process will free all resources and shut down the module.

Prior to the enumeration of the child interfaces the `TLUpdateInterfaceList` function must be called. The list of interfaces held by the System module must not change its content unless this function is called again.

The GenTL Consumer must make sure that calls to the `TLUpdateInterfaceList` function and the functions accessing the list are not made concurrent from multiple threads and that all threads are aware of the update operation, when performed. The GenTL Producer must make sure that any list access is locked either through a reader/writer lock or through a simple mutex.

After the list of available interfaces has been generated internally the `TLGetNumInterfaces` function retrieves the number of present interfaces known to this system. The list contains not the `IF_HANDLE`s itself but their unique IDs of the individual interfaces. To retrieve such an ID the `TLGetInterfaceID` function must be called. This level of indirection allows the enumeration of several interfaces without the need to open them which can save resources.

If additional information is needed to be able to decide which interface is to be opened the `TLGetInterfaceInfo` function can be called. This function enables the GenTL Consumer to query information on a single interface without opening it.

To open a specific interface the unique ID of that interface is passed to the `TLOpenInterface` function. If an ID is known prior to the call this ID can be used to directly open an interface. That implies that the IDs must stay the same in-between two sessions. This is only guaranteed when the hardware does not change in any way. The `TLUpdateInterfaceList` function may be called nevertheless for the creation of the System's internal list of available interfaces. A GenTL Producer may call `TLUpdateInterfaceList` at module instantiation if needed. After successful module instantiation the `TLUpdateInterfaceList` function may only be called by the GenTL Consumer so that it is aware of any change in that list.

For convenience reasons the GenTL Producer implementation may allow opening an Interface module not only using its unique ID but also with any other defined name. If the GenTL Consumer then requests the ID of such a module, the GenTL Producer must return it's unique ID and not the convenience-name used to request the module's handle initially. This allows a GenTL Consumer for example to use the IP address of a network interface (in case of a GigE Vision GenTL Producer driver) to instantiate the module instead of using the unique ID.

When the GenTL Producer driver is not needed anymore the `TLClose` function must be called to close the System module and all other modules which are still open and relate to this System.

After a System module has been closed it may be opened again and the handle to the module may be different from the first instantiation.

3.3 Interface

An Interface module represents a specific hardware interface like a network interface card or a frame grabber. The exact definition of the meaning of an interface is left to the GenTL Producer implementation. After retrieving the `IF_HANDLE` from the System module all attached devices can be enumerated.

If an interface is requested by the GenTL Consumer that has been instantiated before and has not yet been closed, the system must return the handle of the Interface module instance created earlier within the same process. The size and order of the interface list provided by the System module can change during runtime only as a result of a call to the `TLUpdateInterfaceList` function. Interface modules may be closed in a random order that can differ from the order they have been instantiated in. The module does no reference counting. If an Interface module handle is requested twice from within one process space the second call will return an error `GC_ERR_RESOURCE_IN_USE`. The first call from within that process to the `IFClose` function will free all resources and shut down the module in that process.

Every interface is identified not by an index but by a System module wide unique ID. The content of this ID is up to the GenTL Producer and is only interpreted by it and must not be interpreted by the GenTL Consumer.

In order to create or update the internal list of all available devices the `IFUpdateDeviceList` function may be called. The internal list of devices must not change its content unless this function is called again.

The GenTL Consumer must make sure that calls to the `IFUpdateDeviceList` function and the functions accessing the list are not made concurrent from multiple threads and that all threads are aware of an update operation. The GenTL Producer must make sure that any list access is locked either through a reader/writer lock or through a simple mutex.

The number of entries in the internally generated device list can be obtained by calling the `IFGetNumDevices` function. Like the interface list of the System module, this list does not hold the `DEV_HANDLE`s of the devices but their unique IDs. To retrieve an ID from the list call the `IFGetDeviceID` function. By not requiring a device to be opened to be enumerated, it is possible to use different devices in different processes. This is of course only the case if the GenTL Producer supports the access from different processes.

Before opening a Device module more information about it might be necessary. To retrieve that information call the `IFGetDeviceInfo` function.

To open a Device module the IFOpenDevice function is used. As with the interface ID the device ID can be used, if known prior to the call, to open a device directly. The ID must not change between two sessions. The IFUpdateDeviceList function may be called nevertheless for the creation of the Interface internal list of available devices. In case the instantiation of a Device module is possible without having an internal device list the TLOpenDevice may be called without calling IFUpdateDeviceList before. This is necessary if in a system the devices can not be enumerated e.g. a GigE Vision system with a camera connected through a WAN. A GenTL Producer may call IFUpdateDeviceList at module instantiation if needed. After successful module instantiation the IFUpdateDeviceList may only be called by the GenTL Consumer so that it is aware of any change in that list.

For convenience reasons the GenTL Producer implementation may allow to open a Device module not only with its unique ID but with any other defined name. If the GenTL Consumer then requests the ID on such a module, the GenTL Producer must return its unique ID and not the "name" used to request the module's handle initially. This allows a GenTL Consumer for example to use the IP address of a remote device in case of a GigE Vision system to instantiate the Device module instead of using the unique ID.

When an interface is not needed anymore it must be closed with the IFClose function. This frees the resources of this Interface and all child Device modules still open.

After a Interface module has been closed it may be opened again and the handle to the module may be different from the first instantiation.

3.4 Device

A Device module represents the GenTL Producer driver's view on a remote device. If the Device is able to output streaming data this module is used to enumerate the available data streams. The number of available data streams is limited first by the remote device and second by the GenTL Producer implementation. Dependent on the implementation it might be possible that only one of multiple stream channels can be acquired or even only the first one.

If a GenTL Consumer requests a Device that has been instantiated before and has not been closed, the Interface should return an error if the instance was created in another process space. If it explicitly wants to grant access to the Device this access should be read only. If the request comes from the same process space the handle of the instance created earlier must be returned. The module does no reference counting within one process space. If a Device module handle is requested twice from within one process space, the second call will return an error GC_ERR_RESOURCE_IN_USE. The first call from within that process to the DevClose function will free all resources and shut down the module including all child modules in that process.

Every device is identified not by an index but by an Interface module wide unique ID. It is recommended to have a general unique identifier for a specific device. The ID of the GenTL Device module should be different to the remote device ID. The content of this ID is up to the GenTL Producer and is only interpreted by it and not by any GenTL Consumer.

For convenience a GenTL Producer may allow opening a device not only by its unique ID. The other representations may be a user defined name or a transport layer technology dependent ID like for example an IP address for IP-based devices.

To get the number of available data streams the `DevGetNumDataStreams` function is called using the `DEV_HANDLE` returned from the Interface module. It is not intended that the number of data streams changes in one device, but this list is only changed if this function is called again. As with the Interface and the Device lists this list holds the unique IDs of the available streams. The number of data streams may not change during runtime.

To get access to the Port object associated with a Device the function `DevGetPort` must be called.

A Data Stream module can be instantiated by using the `DevOpenDataStream` function. As with the IDs of the modules discussed before a known ID can be used to open a data stream directly. The ID must not change between different sessions. To obtain a unique ID for a Data Stream call the `DevGetDataStreamID` function.

If a device is not needed anymore call the `DevClose` function to free the Device module's resources and its depending child Data Streams if they are still open.

After a Device module has been closed it may be opened again and the handle to the module may be different from the first instantiation.

3.5 Data Stream

The Data Stream module does not enumerate its child modules. Main purpose of this module is the acquisition which is described in detail in chapter 5 Acquisition Engine page 37ff. Buffers are introduced by the calling GenTL Consumer and thus it is not necessary to enumerate them.

Every stream is identified not by an index but by a Device module wide unique ID. The content of this ID is up to the GenTL Producer and is only interpreted by it and not by any GenTL Consumer.

When a Data Stream module is not needed anymore the `DSClose` function must be called to free its resources. This automatically stops a running acquisition, flushes all buffers and revokes them.

If a Data Stream instance is requested by a GenTL Consumer that has been instantiated before and hasn't been closed again, the Device module must return the handle it returned during the earlier instantiation. Access from a different process space is not recommended. The module does no reference counting. That means that even if a Data Stream module handle is requested twice from within one process space the second call will return an error `GC_ERR_RESOURCE_IN_USE`. The first call from within that process to the close function will free all resources and shut down the module in that process.

After a Data Stream module has been closed it may be opened again and the handle to the module may be different from the first instantiation.

3.6 Buffer

A buffer acts as the destination for the data from the acquisition engine.

Every buffer is identified not by an index but by a unique handle returned from the DSAnnounceBuffer or DSAllocAndAnnounceBuffer functions.

A buffer can be allocated either by the GenTL Consumer or by the GenTL Producer. Buffers allocated by the GenTL Consumer are made known to the Data Stream module by a call to DSAnnounceBuffer which returns a BUFFER_HANDLE for this buffer. Buffers allocated by the GenTL Producer are retrieved by a call to DSAllocAndAnnounceBuffer which also returns a BUFFER_HANDLE. The two methods must not be mixed on a single Data Stream module. A GenTL Producer must implement both methods even if one of them is of lesser performance. The simplest implementation for DSAllocAndAnnounceBuffer would be a malloc from the platform SDK.

If DSAnnounceBuffer is called with a pointer to a GenTL Consumer allocated memory which already has been announced before, an error must be returned.

The required size of the buffer must be retrieved either from the Data Stream module the buffer will be announced to or from the associated remote device (see Chapter 5.2.1 for further details).

To allow the acquisition engine to stream data into a buffer it has to be placed into the Input Buffer Pool by calling the DSQueueBuffer function with the BUFFER_HANDLE retrieved through announce functions.

A BUFFER_HANDLE retrieved either by DSAnnounceBuffer or DSAllocAndAnnounceBuffer can be released through a call to DSRevokeBuffer. A buffer which is still in the Input Buffer Pool or the Output Buffer Queue of the acquisition engine cannot be revoked and an error is returned when tried. A memory buffer must only be announced once. If the same buffer is announced twice an error GC_ERR_RESOURCE_IN_USE is returned.

3.7 Example

This sample code shows how to instantiate the first Data Stream of the first Device connected to the first Interface. Start reading through the code in the function ' IFOpenDevice '.

3.7.1 Basic Device access

```
{
  InitLib()
  TL_HANDLE hTL := OpenTL() ;
  IF_HANDLE hIface := OpenFirstInterface(hTL) ;
  DEV_HANDLE hDevice := OpenFirstDevice(hIface) ;
  DS_HANDLE hStream := OpenFirstDataStream(hDevice) ;

    // At this point we have successfully created a data stream on the first
```

```
    // device connected to the first interface. Now we could start to
    // capture data...
  CloseDataStream(hStream)
  CloseDevice(hDevice)
  CloseInterface(hIface)
  CloseTL(hTL)
  CloseLib()
}
```

3.7.2 InitLib

Initialize GenTL Producer

```
{
  GCInitLib()
}
```

3.7.3 GenTL Init

Retrieve TL Handle

```
{
  TLOpen( hTL )
}
```

3.7.4 OpenFirstInterface

Retrieve first Interface Handle

```
{
  TLUpdateInterfaceList ( hTL )
  TLGetNumInterfaces(hTL, NumInterfaces)
  If ( NumInterfaces > 0 )
  {
    // First query the buffer size
    TLGetInterfaceID( hTL, 0, IfaceID, &bufferSize )


    // Open interface with index 0
    TLOpenInterface(hTL, IfaceID, hNewIface)
  }
}
```

3.7.5 OpenFirstDevice

Retrieve first Device Handle

```
{
  IFUpdateDeviceList( hIF );
  IFGetNumDevices( hTL, NumDevices )
  If ( NumDevices > 0 )
  {
    // First query the buffer size
    IFGetDeviceID( hIF, 0, DeviceID, &bufferSize )
```

```
    // Open interface with index 0
    IFOpenDevice( hIF, DeviceID, hNewDevice )
  }
}
```

3.7.6 OpenFirstDataStream

Retrieve first Data Stream

```
{
  // Retrieve the number of Data Stream
  DevGetNumDataStreams( hDev, NumStreams )

  if( NumStreams > 0 )
  {
    // Get ID of first stream using
..  DevGetDataStreamID ( hdev, 0, StreamID, buffersize)
    // Instantiate Data Stream
    DevCreateDataStream (hDev, StreamID, hNewStream )
  }
}
```

3.7.7 CloseDataStream

Close Datastream

```
{
  DSClose(hStream)
}
```

3.7.8 CloseDevice

Close Device

```
{
  DevClose(hDevice)
}
```

3.7.9 CloseInterface

Close Interface

```
{
  IFClose(hIface)
}
```

3.7.10 CloseTL

Close System module

```
{
  TLClose( hTL )
}
```

3.7.11 CloseLib

Shutdown GenTL Producer

```
{
```

```
  GCCloseLib()
}
```

4 Configuration and Signaling

Every module from the System to the Buffer supports a GenTL Port for the Configuration of the module internal settings and the Signaling to the calling GenTL Consumer.

For a detailed description of the C functions interface and data types see chapter 6 Software Interface page 44ff. Before a module can be configured or an event can be registered the module to be accessed must be instantiated. This is done through module enumeration as described in chapter 3 Module Enumeration page 19ff.

4.1 Configuration

To configure a module and access transport layer technology specific settings a GenTL Port with a GenApi compliant XML description is used. The module specific functions concern is the enumeration, instantiation, configuration and basic information retrieval. Configuration is done through a virtual register map and a GenApi XML description for that register map.

For a GenApi reference implementation's IPort interface the TLI publishes Port functions. A GenApi IPort expects a `Read` and a `Write` function which reads a chunk of memory from the associated device. Regarding the GenTL Producer's feature access each module acts as a device for the GenApi implementation by implementing a virtual register map. When certain registers are written or read, implementation dependent operations are performed in the specified module. Thus the abstraction made for camera configuration is transferred also to the GenTL Producer.

The memory layout of that virtual register map is not specified and thus it is up to the GenTL Producer's implementation. A certain set of mandatory features must be implemented which are described in chapter 7, Standard Feature Naming Convention for GenTL page 83ff.

Among the Port functions of the C interface is a `GCReadPort` function and a `GCWritePort` function which can be used to implement an IPort object for the GenApi implementation. These functions resemble the `GCReadPort` and `GCWritePort` functions in their behavior.

4.1.1 Modules

Every GenTL module except the Buffer module must support the Port functions of the TLI – The Buffer module can support these functions. To access the registers of a module the `GCReadPort` and `GCWritePort` functions are called on the module's handle, for example on the `TL_HANDLE` for the System module. A GenApi XML description file and the GenApi Module of GenICam is used to access the virtual register map in the module using GenApi features.

The URL containing the location of the according GenICam XML description can be retrieved through a call to the `GCGetPortURL` function of the C interface.

Additional information about the actual port implementation in the GenTL Producer can be retrieved using `GCGetPortInfo`. The information includes for example the port endianess or the allowed access (read/write, read only,…).

Two modules are special in the way the Port access is handled:

Device Module

In the Device module two ports are available: First the Port functions can be used with a `DEV_HANDLE` giving access to the Device module's internal features. Second the GenTL Consumer can get the `PORT_HANDLE` of the remote device by calling the `DevGetPort` function.

Both Ports are mandatory for a GenTL Producer implementation.

Buffer Module

The implementation of the Port functions is not mandatory for buffers. To check if an implementation is available call the `GCGetPortInfo` function with e.g. the `PORT_INFO_MODULE` command. If no implementation is present the function's return value must be `GC_ERR_NOT_IMPLEMENTED`.

4.1.2 XML Description

The only thing missing to be able to use the GenApi like feature access is the XML description. To retrieve a list with the possible locations of the XML the `GCGetPortURL` function can be called. Three possible locations are defined in a URL like notation (for a definition on the URL see RFC 3986): Module Register Map (recommended), Local Directory or Vendor Web Site

Module Register Map (Recommended)

A URL in the form "local:[///]*filename.extension*;*address*;*length[?SchemaVersion=x.x.x]*" indicates that the XML description file is located in the module's virtual register map. The square brackets are optional. The "x.x.x" stands for the schema version the referenced XML complies to in the form major.minor.subminor. If the SchemaVersion is omitted the URL references to an XML referring to SchemaVersion 1.0.0. Optionally the "///" behind "local:" can be omitted to be compatible to the GigE Vision local format.

If the XML description is stored in the local register map the document can be read by calling the `GCReadPort` function.

Entries in italics must be replaced with actual values as follows:

Table 4-1: Local URL definition for XML description files in the module register map

Entry	Description
Local	Indicates that the XML description file is located in the virtual register map of the module.
filename	Information file name. It is recommended to put the vendor, model/device and revision information in the file name separated by an underscore. For example: tlguru_system_rev1 for the first revision of the System module file of the GenTL Producer company TLGuru.
extension	Indicates the file type. Allowed types are • xml for an uncompressed XML description file. • zip for a zip compressed XML description file.

Entry	Description
address	Start address of the file in the virtual register map. It must be expressed in hexadecimal form without a prefix.
length	Length of the file in bytes. It must be expressed in hexadecimal form without a prefix.
SchemaVersion	Version the referenced XML complies to. The version is specified as a major.minor.subminor.

A complete local URL would look like this:

```
local:tlguru_system_rev1.xml;F0F00000;3BF?SchemaVersion=1.0.0
```

This file has the information file name "tlguru_system_rev1.xml" and is located in the virtual register map starting at address 0xF0F00000 (C style notation) with the length of 0x3BF bytes.

The memory alignment is not further restricted (byte aligned) in a GenTL module. If the platform or the transport layer technology requests a certain memory alignment it has to be taken into account in the GenTL Producer implementation.

Local Directory

URLs in the form "file:///*filepath.extension*" indicate that a file is present somewhere on the machine running the GenTL Consumer. This notation follows the URL definition as in the RFC 3986 for local files. Entries in italics must be replaced with the actual values, for example:

```
file:///C|program%20files/genicam/xml/genapi/tlguru/tlguru_system_rev1.xml?
SchemaVersion=1.0.0
```

This would apply to an uncompressed XML file on an English Microsoft Windows operating system's C drive.

Optionally the "///" behind the "file:" can be omitted to be compatible with the GigE Vision notation. This notation does not specify the exact location. A graphical user interface then would show a file dialog for example.

It is recommended to put the vendor, model or device and revision information in the file name separated by an underscore. For example: tlguru_system_rev1 for the first revision of the System module file of the GenTL Producer company TLGuru.

Supported extensions are:

- xml for uncompressed XML description files
- zip for zip compressed XML description files

Vendor Web Site

If a URL in the form "http://*host*/*path*/*filename.extension*[`?SchemaVersion=1.0.0`]" is present, it indicates that the XML description document can be downloaded from the vendor's web site. This notation follows the URL definition as in the RFC 3986 for the http protocol. Entries in italics must be replaced with the actual values, e.g.

`http://www.tlguru.org/xml/tlguru_system_rev1.xml`

This would apply to an uncompressed XML file found on the web site of the TLGuru company in the xml sub directory.

It is recommended to put the vendor, model or device and revision information in the file name separated by an underscore. For example: tlguru_system_rev1 for the first revision of the System module file of the GenTL Producer company TLGuru.

Supported extensions are:

- xml for uncompressed XML description files
- zip for zip compressed XML description files

4.1.3 Example

```
{
  // Retrieve the URL list
  GCGetPortURL(hModule, URLBuffer, buffersize)

  // Retrieve the a single URL from the list
  // GetSingleURL(URLBuffer, URLString);
  If (ParseURLLocation(URLString) == local)
  {
      // Retrieve the address within the module register map from the URL
      Addr = ParseURLLocalAddress(URLString);
      Length = ParseURLLocalLength(URLString);
      // Retrieve an XMLBuffer to store the XML with the size Length
      …
      // Load xml from local register map into memory
      GCReadPort(hModule, Addr, XMLBuffer, Length )
  }
}
```

4.2 Signaling

The Signaling is used to notify the GenTL Consumer on asynchronous events. Usually all the communication is initiated by the GenTL Consumer. With an event the GenTL Consumer can get notified on specific GenTL Producer operations. This mechanism is an implementation of the Observer Pattern where the calling GenTL Consumer is the observer and the GenTL Producer is being observed.

The reason why an event object approach was chosen rather than callback functions is mainly thread priority problems. A callback function to signal the arrival of a new buffer is normally executed in the thread context of the acquisition engine. Thus all processing in this callback function is done also with its priority. If no additional precautions are taken the acquisition engine is blocked as long the callback function does processing.

By using an event object-based approach for example the acquisition engine implicitly only prepares the necessary data and then signals its availability to the GenTL Consumer through the previously registered event objects. The GenTL Consumer can decide in which thread context and with which priority the data processing is done. Thus processing of the event and the signal's generation are decoupled.

4.2.1 Event Objects

Event objects allow asynchronous signaling to the calling GenTL Consumer.

Event objects have two states: signaled or not signaled. An `EventGetData` function blocks the calling thread until either a user defined timeout occurred or the event object is signaled. If the event object is signaled prior to the call of the `EventGetData` functions, the function returns immediately delivering the data associated with the event signaled.

Not every event type can be registered with every module and not every module needs to implement every possible event type. If a module is not listed for an event it does not have to be implemented in that module's implementation.

The maximum size of the data delivered by an event is defined in the event description and can be retrieved through the `EventGetInfo` function. The actual size is returned by the `EventGetData` function.

If an event type is implemented by a module it is strongly recommended to register an event object for that event type. The following event types are defined:

Table 4-2: Event types per module

Event Type	Modules	Description
Error	All	A GenTL Consumer can get notified on asynchronous errors in a module. These are not errors due to function calls in the C interface or in the GenApi Feature access. These have their own error reporting. This event applies for example to an error while data is acquired in the acquisition engine of a Data Stream module.
New Buffer	Data Stream, Buffer	New data is present in a buffer in the acquisition engine. When registered on a Data Stream module the calling GenTL Consumer is informed about every new buffer in that stream. If it is registered on a Buffer module the GenTL Consumer is notified that this specific buffer contains new data. If the `EventFlush` function is called also all buffers in the output buffer queue are discarded. If a `DSFlushQueue` is called all events from the event queue are removed as well.
Feature Invalidate	Device	This event signals to a calling GenTL Consumer that

Event Type	Modules	Description
	(remote Device)	the GenTL Producer driver changed a value in the register map of the remote device and if this value is cached in the GenApi implementation the cache must be invalidated. This is especially useful with remote devices where the GenTL Producer may change some information that is also used by the GenTL Consumer. For the local modules this is not necessary as the implementation knows which features must not be cached.
Feature Change	Device (remote Device)	This event communicates to a GenTL Consumer that a GenApi feature must be set to a certain value. This is intended for example for the use with the “TLParamsLocked” standard feature. Only the GenTL Producer knows when stream related features must be locked. This event signals the lock ‘1’ or unlock ‘0’ of that feature. The value of a specified feature is changed via its IValue interface, thus a string information is set. No error reporting is done. If that feature is not set or an error occurs no operation is executed and the GenTL Producer is not informed.
Feature Device Event	Device (remote Device)	This event communicates to a calling GenTL Consumer that a GenApi understandable event occurred. The event ID and optional data delivered with this event can be put into a GenApi Adapter which then invalidates all related Nodes.

4.2.2 Event Data Queue

The event data queue is the core of the Signaling. This is a thread safe queue holding event type specific data. Operations on this queue must be locked for example via a mutex in a way that its content may not change when either one of the event functions is accessing it or the module specific thread is accessing it. The GenTL Producer implementation therefore must make sure that access to the queue is as short as possible. Alternatively a lock free queue can be used which supports dequeued operation from multiple threads.

An event object is signaled as long as the event data queue is not empty.

Each event data queue must have its own lock if any to secure the state of each instance and to achieve necessary parallelism. Both read and write operations must be locked. The two operations of event data retrieval and the event object signal state handling in the `EventGetData` function must be atomic. Meaning that, if a lock is used, the lock on the event data queue must be maintained over both operations. Also the operation of putting data in the queue and event object state handling must be atomic.

4.2.3 Event Handling

The handling of the event objects is always the same independent on the event type except for custom events. The signal reason and the signal data of course depend on the event type. The complete state handling for non custom events is done by the GenTL Producer driver. The GenTL Consumer may call the `EventKill` function to terminate a waiting `EventGetData` operation.

The following categories of operations can be differentiated:

Registration

Before the GenTL Consumer can be informed about an event, the event object must be registered. After a module instance has been created in the enumeration process an event object can be created with the `GCRegisterEvent` function. This function returns a unique `EVENT_HANDLE` which identifies the registered event object. To get information about a registered event the `EventGetInfo` function can be used. There must be only one event registered per module and event type. If an event object is registered twice on the same module the `GCRegisterEvent` function must return an error.

To unregister an event object the `GCUnregisterEvent` function must be called. If a module is closed all event registrations are automatically unregistered.

After an `EVENT_HANDLE` is obtained the GenTL Consumer can wait for the event object to be signaled. Upon delivery of an event, the event object carries data. This data is copied into a GenTL Consumer provided buffer when the call to `EventGetData` was successful. The default buffer size, which is always capable of holding all event data, can be queried through the `EventGetInfo` function.

Notification and Data Retrieval

If the event object is signaled data was put into the event data queue at some point in time. The `EventGetData` function can be called to retrieve the actual data. As long as there is only one listener thread this function always returns the stored data. If multiple listener threads are present only one of them returns with the event data while the others stay in a waiting state until either a timeout occurs or until the next event data is available.

When data is read with this function the data is removed from the queue. Afterwards the GenTL Producer implementation checks whether the event data queue is empty or not. If there is more data available the event object stays signaled and next the call to `EventGetData` will deliver the next queue entry. If there are multiple listeners waiting for an event object at the same time, only one of them will return with the data of the next queue entry to the calling GenTL Consumer. Otherwise the event object is reset to not signaled state. The next call of the `EventGetData` function will block until either a timeout occurs or another entry is placed into the event data queue.

The exact type of data is dependent on the event type. The data is copied into a user buffer allocated by the GenTL Consumer. The content of the event data can be queried with the `EventGetDataInfo` function. No data size query must be performed. A call with a `NULL` pointer for the buffer will remove the data from the queue without delivering it. The maximum size of the buffer to be filled is defined by the event type. This information can be queried using the `EventGetInfo` function.

If queued event data is not needed anymore the queue can be emptied by calling the `GCFlushEvent` function on the associated `EVENT_HANDLE`.

Signals that occur without a corresponding event object registered using `GCRegisterEvent` are silently discarded.

4.2.4 Example

This sample shows how to register a NewBuffer event.

```
{
  GCRegisterEvent(hDS, ID_NEW_BUFFER, hNewBufferEvent)
  CreateThread ( AcqFunction )
}
```

AcqFunction

```
{
  while ( !EndRun )
  {
    EventGetData( hNewBufferEvent, EventData )
    If (successful )
    {
      // Do something with the new buffer
    }
  }
}
```

5 Acquisition Engine

5.1 Overview

The acquisition engine is the core of the GenTL data stream. Its task is the transportation itself, which mainly consists of the buffer management.

As stated before the goal for the acquisition engine is to abstract the underlying acquisition mechanism so that it can be used, if not for all, then for most of the acquisition technologies on the market. The target is to acquire data coming from an input stream into memory buffers provided by the GenTL Consumer or made accessible to the GenTL Consumer. The internal design is up to the individual implementation but there are a few directives it has to follow.

As an essential management element a GenTL acquisition engine holds a number of internal logical buffer pools:

5.1.1 Announced Buffer Pool

All announced buffers are referenced here and are thus known to the acquisition engine. A buffer is known from the point when it is announced until it is revoked (removed from the acquisition engine). No buffer may be added to or removed from this pool during acquisition. This also means that a buffer will stay in this pool even when it is delivered to the GenTL Consumer (see below).

The order of the buffers in the pool is not defined. The maximum possible number of buffers in this pool is only limited due to the system resources. The minimum number of buffers in the pool is one or more depending on the technology or the implementation to allow streaming.

5.1.2 Input Buffer Pool

When the acquisition engine receives data from a device it will fill a buffer from this pool. While a buffer is filled it is removed from the pool and if successful put into the output buffer queue. If the transfer was not successful the buffer remains in the input pool by default.

The order of the buffers in the pool is not defined. Only buffers present in the Announced Buffer Pool can be in this pool. The maximum number of buffers in this pool is the number of announced buffers.

5.1.3 Output Buffer Queue

Once a buffer has been successfully filled, it is placed into this queue. As soon as there is at least one buffer in the output buffer queue a previous registered event object gets signaled and the GenTL Consumer can retrieve the event data and thus can identify the filled buffer.

When the event data is retrieved the associated buffer is removed from the output buffer queue. This also means that the data and thus the buffer can only be retrieved once. After the buffer is removed from the output buffer queue (delivered) the acquisition engine must not write data into it. Thus this effectively a buffer locking mechanism.

In order to reuse this buffer a GenTL Consumer has to put the buffer back into the Input Buffer Pool (requeue).

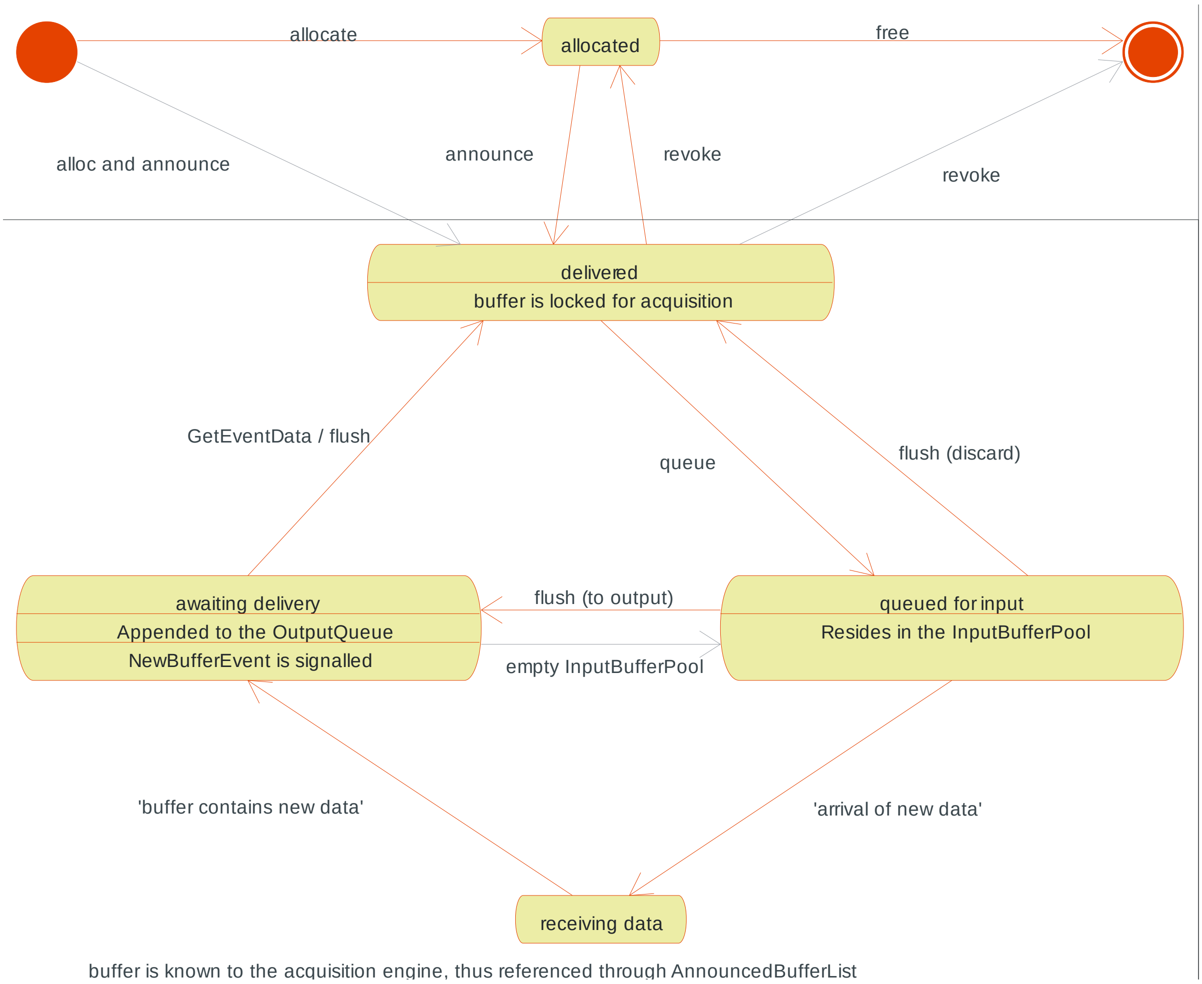
The order of the buffers is defined by the acquisition mode. Buffers are retrieved by the “New Buffer” event in a logical first-in-first-out manner. If the acquisition engine does not remove or reorder buffers in the Output Buffer Queue (see the different acquisition modes in the GenICam Standard Feature Naming Convention), it is always the oldest buffer from the queue that is returned to the GenTL Consumer. Only buffers present in the Announced Buffer Pool which were filled can be in this queue.

5.2 Acquisition Chain

The following description shows the steps to acquire an image from the GenTL Consumer’s point of view. Image or data acquisition is performed on the Data Stream module with the functions using the `DS_HANDLE`. Thus before an acquisition can carried out, an enumeration of a Data Stream module has to be performed (see chapter 3 Module Enumeration page 19ff). For a detailed description of the C functions and data types see chapter 6 Software Interface page 44ff.

Prior to the following steps the remote device and if necessary (in case a grabber is used) the GenTL Device module, should be configured to produce the desired image format. The remote device’s `PORT_HANDLE` can be retrieved from the GenTL Device module’s `DevGetPort` function.

Figure 5-1: Acquisition chain seen from a buffer’s perspective (default acquisition mode)



5.2.1 Allocate Memory

First the size of a single buffer has to be obtained. This information is first queried from the GenTL Data Stream module (important: not from the remote device). Check if the standard feature “PayloadSize” is present or if the `DSGetInfo` with the command `STREAM_INFO_DEFINES_PAYLOADSIZE` returns true. If not, query the information from the remote device features. The remote device port can be retrieved via the `DevGetPort` function from the according Device module. The GenTL Comsumer has to select the right streaming channel in the remote device and read the according “PayloadSize” standard feature.

With that information one or multiple buffers can be allocated as the GenTL Consumer sees fit. The allocation can also be done by the GenTL Producer driver with the combined `DSAllocAndAnnounceBuffer` function. If the buffers are larger than requested it does not matter and the real size can be obtained through the `DSGetBufferInfo` function. If the buffers are smaller than requested the error event is fired on the buffer (if implemented) and on the transmitting data stream and the buffer may only be partly filled.

5.2.2 Announce Buffers

All buffers to be used in the acquisition engine must be made known prior to their use. Buffers can be added (announced) and removed (revoked) at any time no grab is active. Along with the buffer memory a pointer to user data is passed which may point to a buffer specific implementation. That pointer is delivered along with the Buffer module handle in the "New Buffer" event.

The `DSAnnounceBuffer` and `DSAllocAndAnnounceBuffer` functions return a unique `BUFFER_HANDLE` to identify the buffer in the process. The minimum number of buffers that must be announced depends on the technology used. This information can be queried from the Data Stream module features. If there is a known maximum this can also be queried. Otherwise the number of buffers is only limited by available memory.

The acquisition engine normally stores additional data with the announced buffers to be able to e.g. use DMA transfer to fill the buffers.

5.2.3 Queue Buffers

To acquire data at least one buffer has to be queued with the `DSQueueBuffer` function. When a buffer is queued it is put into the Input Buffer Pool. The user has to explicitly call `DSQueueBuffer` to place the buffers into the Input Buffer Pool. The order in which the buffers are queued does not need to match the order in which they were announced. Also the queue order does not necessarily have an influence in which order the buffers are filled. This depends only on the acquisition mode.

5.2.4 Register "New Buffer" Event

An event object to the data stream must be registered using the `NewBufferEvent` ID in order to be notified on newly filled buffers. The `GCRegisterEvent` function returns a unique `EVENT_HANDLE` which can be used to obtain event specific data when the event was signaled. For the "New Buffer" event this data is the `BUFFER_HANDLE` and the user data pointer.

5.2.5 Start Acquisition

First the acquisition engine on the host is started with the `DSStartAcquisition` function. After that start the acquisition on the device if necessary by querying the "StartAcquisition" standard feature on the remote device via the GenICam GenApi. If the feature is present execute the command.

5.2.6 Acquire Image Data

This action is performed in a loop:

- Wait for the "New Buffer" event to be signaled (see 4.2 Signaling page 32ff)
- Process image data
- Requeue buffer in the Input Buffer Pool

With the event data from the signaled event the newly filled buffer can be obtained and then processed. As stated before no assumptions on the order of the buffers are made except that the acquisition mode defines it.

Requeuing the buffers can be done in any order with the `DSQueueBuffer` function. As long as the buffer is not in the Input Buffer Pool or in the Output Buffer Queue the acquisition engine will not write into the buffer. Thus the buffer is effectively locked.

5.2.7 Stop Acquisition

When finished acquiring image data first stop the acquisition on the remote device if necessary. This can be done by querying the "StopAcquisition" standard feature on the remote device. If it is present the command should be executed. Afterwards the `DSStopAcquisition` function is called to stop the acquisition on the host. By doing that the status of the buffers does not change. That implies that a buffer that is in the Input Buffer Pool remains there. This is the same for buffers in the Output Buffer Queue. This has the advantage that buffers which were filled during the acquisition stop process still can be retrieved and processed.

5.2.8 Flush Buffer Pools and Queues

In order to clear the state of the buffers to allow revoking them, the buffers have to be flushed either with the `DSFlushQueue` function or with the `EventFlush` function. With the `DSFlushQueue` function buffers from the Input Buffer Pool can either be flushed to the Output Buffer Queue or discarded. Buffers from the Output Buffer Queue also must either be processed or flushed. Flushing the Output Buffer Queue is done by calling `EventFlush` function. Using the `EventFlush` function on the "New Buffer" event the buffers from the Output Buffer Queue are discarded.

5.2.9 Revoke Buffers

To enable the acquisition engine to free all resources needed for acquiring image data revoke the announced buffers. Buffers that are referenced in either the Input Buffer Pool or the Output Buffer Queue can not be revoked. After revoking a buffer with the `DSRevokeBuffer` function it is not known to the acquisition engine and thus can neither be queued nor receive any image data.

The order in which buffers can be revoked depends on the method in which they where announced. Buffers can be revoked in any order if they were announced by the `DSAnnounceBuffer` function. More care has to be taken if the `DSAllocAndAnnounceBuffer` function is used. Normally underlying acquisition engines must not change the base pointer to the memory containing the data within a buffer object. If the `DSAllocAndAnnounceBuffer` function is used the base pointer of a buffer object may change after another buffer object has been revoked using the `DSRevokeBuffer` function.

5.2.10 Free Memory

If the GenTL Consumer provided the memory for the buffers using the `DSAnnounceBuffer` function it also has to free it. Memory allocated by the GenTL

Producer implementation with `DSAllocAndAnnounceBuffer` function is freed by the library if necessary. The GenTL Consumer must not free this memory.

5.3 Acquisition Modes

Acquisition modes describe the internal buffer handling during acquisition. There is only one mandatory default mode. More acquisition modes are defined in the GenICam Standard Feature Naming Convention document.

A certain mode may differ from another in these aspects:

- Which buffer is taken from the Input Buffer Pool to be filled
- At which time a filled buffer is moved to the Output Buffer Queue and at which position it is inserted
- Which buffer in the Output Buffer Queue is overwritten (if any at all) on an empty Input Buffer Pool

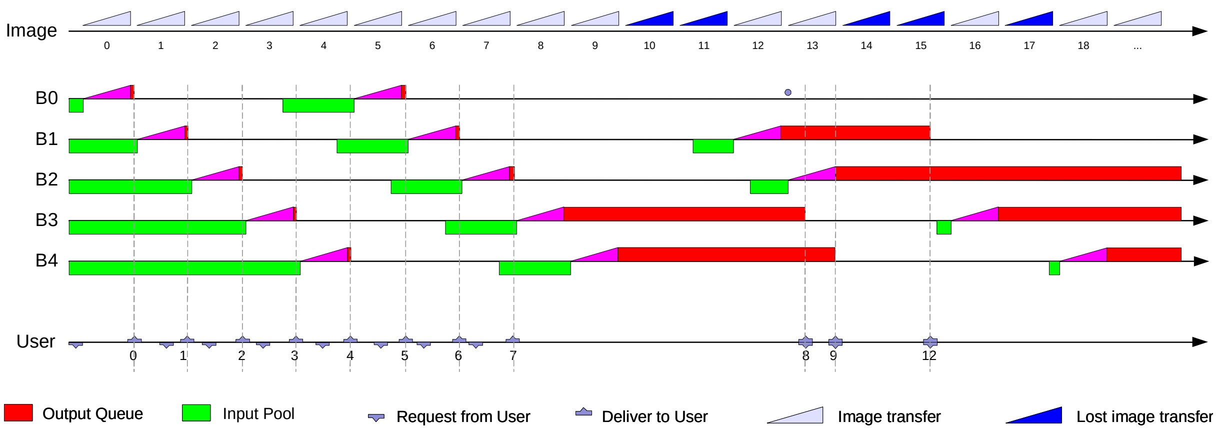
The graphical description assumes that we are looking on an acquisition start scenario with five announced and queued buffers B0 to B4 ready for acquisition. When a buffer is delivered the image number is stated below that event. A solid bar on a buffer's time line illustrates its presence in a Buffer pool. A ramp indicates image transfer and therefore transition. During a thin line the Buffer is controlled by the GenTL Consumer and locked for data reception.

5.3.1 Default Mode

The default mode is designed to be simple and flexible with only a few restrictions. This is done to be able to map it to most acquisition techniques used today. If a specific technique can not be mapped to this mode the goal has to be achieved by copying the data and emulating the behavior in software.

In this scenario every acquired image is delivered to the GenTL Consumer if the mean processing time is below the acquisition time. Peaks in processing time can be mitigated by a larger number of buffers.

Figure 5-2: Default acquisition from the GenTL Consumer's perspective



The buffer acquired first (the oldest) is always delivered to the GenTL Consumer. No buffer is discarded or overwritten in the Output Buffer Queue. By successive calls to retrieve the event data (and thus the buffers) all filled buffers are delivered in the order they were acquired. This is done regardless of the time the buffer was filled.

It is not defined which buffer is taken from the Input Buffer Pool if new image data is received. If no buffer is in the Input Buffer Pool (e.g. the requeuing rate falls behind the transfer rate over a sufficient amount of time), an incoming image will be lost. The acquisition engine will be stalled until a buffer is requeued.

Wrap-Up:

- There’s no defined order in which the buffers are taken from the Input Buffer Pool.
- As soon as it is filled a buffer is placed at the end of the Output Buffer Queue.
- The acquisition engine stalls if the Input Buffer Pool becomes empty and as long as a buffer is queued.

6 Software Interface

6.1 Overview

A GenTL Producer implementation is provided as a platform dependent dynamic loadable library; under Microsoft Windows platform this would be a dynamic link library (DLL). The file extension of the library is 'cti' for "Common Transport Interface". To enable easy dynamical loading and to support a wide range of languages a C interface is defined. It is designed to be minimal and complete regarding enumeration and the access to Configuration and Signaling. This enables a quick implementation and reduces the workload on testing.

All functions defined in this chapter are mandatory and must be implemented and exported in the libraries interface; even if no implementation for a function is necessary.

6.1.1 Function Naming Convention

All functions of the TLI follow a common naming scheme:

Prefix Operation Specifier

Entries in italics are replaced by an actual value as follows:

Table 6-1: Function naming convention

Entry	Description
Prefix	Specifies the handle the function works on. The handle represents the module used. Values: • GC if applicable for no or all modules (GC for GenICam) • TL for System module (TL for Transport Layer) • IF for Interface module (IF for Interface) • Dev for Device module (Dev for Device) • DS for Data Stream module (DS for Data Stream) • Event for Event Objects
Operation	Specifies the operation done on a certain module. Values (choice): • Open to open a module • Close to close a module • Get to query information about a module or object
Specifier	This is optional. If an operation needs additional information, it is provided by the *Specifier*. Values (choice): • xxxInfo to retrieve xxx-object specific information • Numxxx to retrieve the number of xxx-objects

For example the function `TLGetNumInterfaces` works on the System module's `TL_HANDLE` and queries the number of available interfaces. `TLClose` for instance closes the System module.

6.1.2 Memory and Object Management

The interface is designed in a way that objects and data allocated in the GenTL Producer implementation are only freed or changed inside the library. Vice versa all objects and data allocated by the calling GenTL Consumer must only be changed and freed by the calling GenTL Consumer. No language specific features except the ones allowed by ANSII C and published in the interface are allowed.

The function calling convention is stdcall where applicable. If this calling convention is not possible on an architecture (e.g. on a non x86 architecture) and multiple conventions are possible they must be defined.

This ensures that the GenTL Producer implementation and the calling GenTL Consumer can use different heaps and different memory allocation strategies. Also language interchangeability is easier handled this way.

For functions filling a buffer (e.g. a C string) the function can be called with a `NULL` pointer for the `char*` parameter (buffer). The *piSize* parameter is then filled with the size of buffer needed to hold the information in bytes. For C strings that does incorporate the terminating 0 character. A function expecting a C string as its parameter not containing a size parameter for it expects a 0-terminated C string. Queries are not allowed for event data.

Objects that contain the state of one module's instance are referenced by handles (`void*`). If a module has been instantiated before and is opened again, the already existing handle has to be returned. A close on the module will free the resource of the closed module and all underlying or depending child modules. This is true as long as these calls are in the same process space (see below). Thus if a Interface module is closed all attached Device, Data Stream and Buffer modules are also closed.

6.1.3 Thread and Multiprocess Safety

If the platform supports threading, all functions must be thread safe to ensure data integrity when a function is called from different threads in one process. If the platform supports independent processes the GenTL Producer implementation must establish interprocess communication. At minimum other processes are not allowed to use an opened System module. It is recommended though that a GenTL Producer implementation is multi process capable to the point where:

- Access rights for the Modules are checked
  An open Device module should be locked against multiple process access. In that case an error should be returned.
- Data/state safety is ensured
  Reference counting must be done so that if e.g. the System module of one process is closed the resources of another process are not automatically freed.
- Different processes can communicate with different devices
  Each process should be able to communicate with one or multiple devices. Also different processes should be able to communicate with different devices.

6.1.4 Error Handling

Every function has as its return value a `GC_ERROR`. This value indicates the status of the operation. Functions must give strong exception safety. With an exception not a language dependent exception object is meant, but an execution error in the called function with a return code other than `GC_ERR_SUCCESS`. No exception objects may be thrown of any exported function. Strong exception safety means:

- Data validity is preserved
  No data becomes corrupted or leaked.
- State is unchanged
  First the internal state must stay consistent and it must be as if the function encountering the error was never called.

This ensures that calling GenTL Consumers always can expect a known state in the GenTL Producer implementation: either it is the desired state when a function call was successful or it is the state the GenTL Producer implementation had before the call.

The following values are defined:

Table 6-2: C interface error codes

Enumerator	Value	Description
`GC_ERR_SUCCESS`	0	Operation was successful; no error occurred.
`GC_ERR_ERROR`	-1001	Unspecified runtime error.
`GC_ERR_NOT_INITIALIZED`	-1002	Module not initialized; e.g. `GCInitLib` was not called.
`GC_ERR_NOT_IMPLEMENTED`	-1003	Requested operation not implemented; e.g. no Port functions on a Buffer module.
`GC_ERR_RESOURCE_IN_USE`	-1004	Requested module is used; e.g. in another process.
`GC_ERR_ACCESS_DENIED`	-1005	Requested operation is not allowed; e.g. a remote device is opened by another client.
`GC_ERR_INVALID_HANDLE`	-1006	Given handle does not support the operation; e.g. function call on wrong handle or `NULL` pointer.
`GC_ERR_INVALID_ID`	-1007	ID could not be connected to a resource; e.g. a device with the given ID is currently not available.
`GC_ERR_NO_DATA`	-1008	The function has no data to work on; e.g. the `GCGetEventData` function was called on an empty event data queue.
`GC_ERR_INVALID_PARAMETER`	-1009	One of the parameter given was not valid or out of range and none of the

Enumerator	Value	Description
		error codes above fits.
GC_ERR_IO	-1010	Communication error has occurred; e.g. a read or write operation to a remote device failed.
GC_ERR_TIMEOUT	-1011	An operation's timeout time expired before it could be completed.
GC_ERR_CUSTOM_ID	-10000	Any error smaller or equal than –10000 is implementation specific

To get a detailed descriptive text about the error reason call the GCGetLastError function.

This section contains all definitions valid for the whole C interface and functions bound only to the library itself.

6.2 Used Data Types

To have a defined stack layout certain data types have a primitive data type as its base.

GC_ERROR

The return value of all functions is a 32 bit signed integer value.

Handles

All handles like TL_HANDLE or PORT_HANDLE are void*. The size is platform dependent (e.g. 32 bit on 32 bit platforms)

Enumerations

All enumerations are of type enum. In order to allow implementation specific extensions all enums are accompanied by a 32 bit integer value. On platforms/compilers where this is not the case a primitive data type with a matching size is to be used.

Buffers and C Strings

Buffers are normally typed as void* if arbitrary data is accessed. For specialized buffers like for C strings a char* is used. A char is expected to have 8 bit. On platforms/compilers where this is not the case a byte like primitive data type must be used.

String encoding is ASCII (characters with numerical values up to and including 127). A string as an input value without a size parameter must be 0-terminated. Strings with a size parameter must include the terminating 0.

Primitive Data Types

The size_t type indicates that a buffer size is represented. This is a platform dependent unsigned integer (e.g. 32 bit on 32 bit platforms).

The other functions use a notation defining its base type and size. uint8_t stands for an unsigned integer with the size of 8 bits. int32_t is a signed integer with 32 bits size.

6.3 Function Declarations

6.3.1 Library Functions

```
GC_ERROR  GCCloseLib             ( void )
```

This function must be called after no function of the GenTL library is needed anymore to clean up the resources from the `GCInitLib` function call. Multiple calls should be ignored.

`GCGetLastError` must not be called after the call of this function!

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

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

Inquire information about a GenTL implementation without opening the TL. The available information is limited since the TL is not initialized yet. Even if this function works on a closed library, `GCInitLib` must be called prior calling this function.

If the provided buffer is too small to receive all information an error is returned.

Parameters

[in]	*iInfoCmd*	Information to be retrieved as defined in `TL_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `TL_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *iType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  GCGetLastError      ( GC_ERROR *      piErrorCode,
                                char *          sErrorText,
                                size_t *        piSize )
```

Returns a readable text description of the last error occurred in the local thread context.

If multiple threads are supported on a platform this function must store this information thread local.

Parameters

[out]	*piErrorCode*	Error code of the last error.
[in,out]	*sErrorText*	Pointer to a user allocated C string buffer to receive the last error text. If this parameter is `NULL`, *piSize* will contain the needed size of *sErrorText* in bytes. The size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL`: out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL`: in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  GCInitLib           ( void )
```

This function must be called prior to any other function call to allow global initialization of the GenTL Producer driver. This function is necessary since automated initialization functionality like within `DllMain` on MS Windows platforms is very limited. Multiple calls should be ignored.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

6.3.2 System Functions

```
GC_ERROR  TLClose             ( TL_HANDLE hSystem )
```

Closes the System module associated with the given *hSystem* handle. This closes the whole GenTL Producer driver and frees all resources. Call the `GCCloseLib` function afterwards if the library is not needed anymore.

Parameters

[in]	*hSystem*	System module handle to close.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLGetInfo            ( TL_HANDLE       hSystem,
                                 TL_INFO_CMD     iInfoCmd,
                                 INFO_DATATYPE * piType,
                                 void *          pBuffer,
                                 size_t *        piSize )
```

Inquire information about the System module as defined in `TL_INFO_CMD`.

Parameters

[in]	*hSystem*	System module to work on.
[in]	*iInfoCmd*	Information to be retrieved as defined in `TL_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `TL_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLGetInterfaceID     ( TL_HANDLE       hSystem,
                                 uint32_t        iIndex,
                                 char *          sIfaceID,
                                 size_t *        piSize )
```

Queries the unique ID of the interface at *iIndex* in the internal interface list. Prior to this call the `TLUpdateInterfaceList` function must be called. The list content will not change until the next call of the update function.

Parameters

[in]	*hSystem*	System module to work on.
[in]	*iIndex*	Zero-based index of the interface on this system.
[in,out]	*sIfaceID*	Pointer to a user allocated C string buffer to receive the Interface module ID at the given *iIndex*. If this parameter is

		NULL, *piSize* will contain the needed size of *sIfaceID* in bytes. The size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal NULL: out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal NULL: in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLGetInterfaceInfo  ( TL_HANDLE         hSystem,
                                const char *      sIfaceID,
                                INTERFACE_INFO_CMD iInfoCmd,
                                INFO_DATATYPE *   piType,
                                void *            pBuffer,
                                size_t *          piSize )
```

Inquire information about an interface on the given System module *hSystem* as defined in INTERFACE_INFO_CMD without opening the interface.

Parameters

[in]	*hSystem*	System module to work on.
[in]	*sIfaceID*	Unique ID of the interface to inquire information from.
[in]	*iInfoCmd*	Information to be retrieved as defined in INTERFACE_INFO_CMD.
[out]	*piType*	Data type of the *pBuffer* content as defined in the INTERFACE_INFO_CMD and INFO_DATATYPE.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal NULL: out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal NULL: in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLGetNumInterfaces  ( TL_HANDLE       hSystem,
                                uint32_t *      piNumIfaces )
```

Queries the number of available interfaces on this System module. Prior to this call the `TLUpdateInterfaceList` function must be called. The list content will not change until the next call of the update function.

Parameters

[in]	*hSystem*	System module to work on.
[out]	*piNumIfaces*	Number of interfaces on this System module.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLOpen              ( TL_HANDLE * phSystem )
```

Opens the System module and puts the instance in the *phSystem* handle. This allocates all system wide resources. Call the `GCInitLib` function before this function. A System module can only be opened once.

Parameters

[out]	*phSystem*	System module handle of the newly opened system.

Returns

`GC_ERROR`: `GC_ERR_RESOURCE_IN_USE` if the module is currently open.

Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLOpenInterface     ( TL_HANDLE       hSystem,
                                const char *    sIfaceID,
                                IF_HANDLE *     phIface )
```

Opens the given *sIfaceID* on the given *hSystem*. Prior to this call the `TLUpdateInterfaceList` function must be called.

Any subsequent call to `TLOpenInterface` with an *sIfaceID* which has already been opened will return an error `GC_ERR_RESOURCE_IN_USE`.

The interface ID need not match the one returned from `TLGetInterfaceID`. As long as the GenTL Producer knows how to interpret that ID it will return a valid handle. For example, if in a specific implementation the interface has a user-defined name, this function will return a valid handle as long as the provided name refers to an internally known interface.

Parameters

[in]	*hSystem*	System module to work on.
[in]	*sIfaceID*	Unique interface ID to open as a 0-terminated C string.
[out]	*phIface*	Interface handle of the newly created interface.

Returns

GC_ERROR: GC_ERR_RESOURCE_IN_USE if the module is currently open.

Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  TLUpdateInterfaceList ( TL_HANDLE    hSystem,
                                  bool8_t *    pbChanged,
                                  uint64_t     iTimeout )
```

Updates the internal list of available interfaces. This may change the connection between a list index and an interface ID.

Parameters

[in]	*hSystem*	System module to work on.
[out]	*pbChanged*	Contains true if the internal list was changed and false otherwise. If set to NULL nothing is written to this parameter.
[in]	*iTimeout*	Timeout in ms. If set to 0xFFFFFFFFFFFFFFFF the timeout is infinite.

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

6.3.3 Interface Functions

```
GC_ERROR  IFClose              ( IF_HANDLE    hIface )
```

Closes the Interface module associated with the given *hIface* handle. This closes all dependent Device modules and frees all interface related resources.

Parameters

[in]	*hSystem*	System module handle to close.

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  IFGetInfo            ( IF_HANDLE         hIface,
                                 INTERFACE_INFO_CMD iInfoCmd,
                                 INFO_DATATYPE *   piType,
                                 void *            pBuffer,
                                 size_t *          piSize )
```

Inquire information about the Interface module as defined in INTERFACE_INFO_CMD.

Parameters

[in]	*hIface*	Interface module to work on.
[in]	*iInfoCmd*	Information to be retrieved as defined in INTERFACE_INFO_CMD.

[out]	*piType*	Data type of the *pBuffer* content as defined in the `INTERFACE_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  IFGetDeviceID        ( IF_HANDLE       hIface,
                                 uint32_t        iIndex,
                                 char *          sDeviceID,
                                 size_t *        piSize )
```

Queries the unique ID of the device at *iIndex* in the internal device list. Prior to this call the `IFUpdateDeviceList` function must be called. The list content will not change until the next call of the update function.

Parameters

[in]	*hIface*	Interface module to work on.
[in]	*iIndex*	Zero-based index of the device on this interface.
[in,out]	*sDeviceID*	Pointer to a user allocated C string buffer to receive the Device module ID at the given *iIndex*. If this parameter is `NULL`, *piSize* will contain the needed size of *sDeviceID* in bytes. The size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  IFGetDeviceInfo    ( IF_HANDLE        hIface,
                               const char *     sDeviceID,
                               DEVICE_INFO_CMD  iInfoCmd,
                               INFO_DATATYPE *  piType,
                               void *           pBuffer,
                               size_t *         piSize )
```

Inquire information about a device on the given Interface module *hIface* as defined in `DEVICE_INFO_CMD` without opening the device.

Parameters

[in]	*hIface*	Interface module to work on.
[in]	*sDeviceID*	Unique ID of the device to inquire information about.
[in]	*iInfoCmd*	Information to be retrieved as defined in `DEVICE_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `DEVICE_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  IFGetNumDevices    ( IF_HANDLE        hIface,
                               uint32_t *       piNumDevices )
```

Queries the number of available devices on this Interface module. Prior to this call the `IFUpdateDeviceList` function must be called. The list content will not change until the next call of the update function.

Parameters

[in]	*hIface*	Interface module to work on.
[out]	*piNumDevices*	Number of devices on this Interface module.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  IFOpenDevice       ( IF_HANDLE      hIface,
                               const char *   sDeviceID,
                               DEVICE_ACCESS_FLAGS iOpenFlags,
                               DEV_HANDLE * phDevice )
```

Opens the given *sDeviceID* with the given *iOpenFlags* on the given *hIface*. Prior to this call the `IFUpdateDeviceList` function must be called.

Any subsequent call to `IFOpenDevice` with an *sDeviceID* which has already been opened will return an error `GC_ERR_RESOURCE_IN_USE`.

The device ID need not match the one returned from `IFGetDeviceID`. As long as the GenTL Producer knows how to interpret that ID it will return a valid handle. For example, if in a specific implementation the device has a user-defined name, this function will return a valid handle as long as the provided name refers to an internally known device.

Parameters

[in]	*hIface*	Interface module to work on.
[in]	*sDeviceID*	Unique device ID to open as a 0-terminated C string.
[in]	*iOpenFlags*	Configures the open process as defined in the `DEVICE_ACCESS_FLAGS`.
[out]	*phDevice*	Device handle of the newly created Device module.

Returns

`GC_ERROR`: `GC_ERR_RESOURCE_IN_USE` if the module is currently open.

Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  IFUpdateDeviceList ( IF_HANDLE      hIface,
                               bool8_t *      pbChanged,
                               uint64_t       iTimeout )
```

Updates the internal list of available devices. This may change the connection between a list index and a device ID.

Parameters

[in]	*hIface*	Interface module to work on.
[out]	*pbChanged*	Contains `true` if the internal list was changed and `false` otherwise. If set to `NULL` nothing is written to this parameter.
[in]	*iTimeout*	Timeout in ms. If set to 0xFFFFFFFFFFFFFFFF the timeout is infinite.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

6.3.4 Device Functions

```
GC_ERROR  DevClose              ( DEV_HANDLE hDevice )
```

Closes the Device module associated with the given *hDevice* handle. This frees all resources of the Device module and closes all dependent Data Stream module instances.

Parameters

[in]	*hDevice*	Device module handle to close.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DevGetInfo            ( DEV_HANDLE      hDevice,
                                  DEVICE_INFO_CMD iInfoCmd,
                                  INFO_DATATYPE * piType,
                                  void *          pBuffer,
                                  size_t *        piSize )
```

Inquire information about the Device module as defined in `DEVICE_INFO_CMD`.

Parameters

[in]	*hDevice*	Device module to work on.
[in]	*iInfoCmd*	Information to be retrieved as defined in `DEVICE_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `DEVICE_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DevGetDataStreamID  ( DEV_HANDLE      hDevice,
                                uint32_t        iIndex,
                                char *          sDataStreamID,
                                size_t *        piSize )
```

Queries the unique ID of the data stream at *iIndex* in the internal data stream list.

Parameters

[in]	*hDevice*	Device module to work on.
[in]	*iIndex*	Zero-based index of the data stream on this device.
[in,out]	*sDataStreamID*	Pointer to a user allocated C string buffer to receive the Interface module ID at the given *iIndex*. If this parameter is `NULL`, *piSize* will contain the needed size of *sDataStreamID* in bytes. The size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DevGetNumDataStreams( DEV_HANDLE     hDevice,
                                uint32_t *piNumDataStreams)
```

Queries the number of available data streams on this Device module.

Parameters

[in]	*hDevice*	Device module to work on.
[out]	*piNumDataStreams*	Number of data stream on this Device module.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DevGetPort          ( DEV_HANDLE      hDevice,
                                PORT_HANDLE *   phRemoteDev )
```

Retrieves the port handle for the associated remote device.

This function does not return the handle for the Port functions for the Device module but for the physical remote device.

The *phRemoteDev* handle must not be closed explicitly. This is done automatically when `DevClose` is called on this Device module.

Parameters

[in]	*hDevice*	Device module to work on.
[out]	*phRemoteDev*	Port handle for the remote device.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DevOpenDataStream   ( DEV_HANDLE      hDevice,
                                const char *    sDataStreamID,
                                DS_HANDLE *     phDataStream )
```

Opens the given *sDataStreamID* on the given *hDevice*.

Any subsequent call to `DevOpenDataStream` with an *sDataStreamID* which has already been opened will return an error `GC_ERR_RESOURCE_IN_USE`.

The Data Stream ID need not match the one returned from `DevGetDataStreamID`. As long as the GenTL Producer knows how to interpret that ID it will return a valid handle. For example, if in a specific implementation the data stream has a user defined name, this function will return a valid handle as long as the provided name refers to an internally known data stream.

Parameters

[in]	*hDevice*	Device module to work on.
[in]	*sDataStreamID*	Unique data stream ID to open as a 0-terminated C string.
[out]	*phDataStream*	Data Stream module handle of the newly created stream.

Returns

`GC_ERROR`: `GC_ERR_RESOURCE_IN_USE` if the module is currently open.

Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

6.3.5 Data Stream Functions

```
GC_ERROR  DSAllocAndAnnounceBuffer ( DS_HANDLE hDataStream,
                                     size_t          iBufferSize,
                                     void *          pPrivate,
                                     BUFFER_HANDLE * phBuffer )
```

This function allocates the memory for a single buffer and announces this buffer to the Data Stream associated with the *hDataStream* handle and returns a *hBuffer* handle which references that single buffer until the buffer is revoked. This will allocate internal resources which will be freed upon a call to `DSRevokeBuffer`.

Announcing a buffer to a data stream does not mean that this buffer will be automatically queued for acquisition. This is done through a separate call to `DSQueueBuffer`.

The memory referenced in this buffer must stay valid until a buffer is revoked with `DSRevokeBuffer`.

Every call of this function must be matched with a call of `DSRevokeBuffer`.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*iBufferSize*	Size of the buffer in bytes.

[in]	*pPrivate*	Pointer to private data which will be passed to the GenTL Consumer on `NewBuffer` events.
[out]	*phBuffer*	Buffer module handle of the newly announced buffer.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSAnnounceBuffer    ( DS_HANDLE       hDataStream,
                                void *          pBuffer,
                                size_t          iSize,
                                void *          pPrivate,
                                BUFFER_HANDLE * phBuffer)
```

This announces a GenTL Consumer allocated memory to the Data Stream associated with the *hDataStream* handle and returns a *hBuffer* handle which references that single buffer until the buffer is revoked. This will allocate internal resources which will be freed upon a call to `DSRevokeBuffer`.

Announcing a buffer to a data stream does not mean that this buffer will be automatically queued for acquisition. This is done through a separate call to `DSQueueBuffer`.

The memory referenced in *pBuffer* must stay valid until the buffer is revoked with `DSRevokeBuffer`. Every call of this function must be matched with a call of `DSRevokeBuffer`.

A buffer can only be announced once. If a GenTL Consumer tries to announce an already announced buffer the function will return an error `GC_ERR_RESOURCE_IN_USE`.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*pBuffer*	Pointer to buffer memory to announce.
[in]	*iSize*	Size of the *pBuffer* in bytes.
[in]	*pPrivate*	Pointer to private data which will be passed to the GenTL Consumer on `NewBuffer` events.
[out]	*phBuffer*	Buffer module handle of the newly announced buffer.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSClose             ( DS_HANDLE hDataStream )
```

Closes the Data Stream module associated with the given *hDataStream* handle. This frees all resources of the Data Stream module, discards and revokes all buffers.

Parameters

[in] *hDataStream* Data Stream module handle to close.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSFlushQueue        ( DS_HANDLE       hDataStream,
                                ACQ_QUEUE_TYPE  iOperation )
```

Flushes the by *iOperation* defined internal buffer pool or queue to another one as defined in `ACQ_QUEUE_TYPE`.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*iOperation*	Flush operation to perform as defined in `ACQ_QUEUE_TYPE`.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSGetBufferID       ( DS_HANDLE       hDataStream,
                                uint32_t        iIndex,
                                BUFFER_HANDLE * phBuffer )
```

Queries the buffer handle for a given iIndex. The buffer handle works as a unique ID of the Buffer module.

Note that the relation between index and buffer handle might change with additional announced and/or revoked buffers. As long as no buffers are announced or revoked this relation must not change.

The number of announced buffers can be queried with the `DSGetInfo` function.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*iIndex*	Zero-based index of the buffer on this data stream.
[in,out]	*phBuffer*	Buffer module handle of the given *iIndex*.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSGetBufferInfo     ( DS_HANDLE        hDataStream,
                                BUFFER_HANDLE    hBuffer,
                                BUFFER_INFO_CMD  iInfoCmd,
                                INFO_DATATYPE *  piType,
                                void *           pBuffer,
                                size_t *         piSize )
```

Inquire information about the Buffer module associated with *hBuffer* on the *hDataStream* instance as defined in `BUFFER_INFO_CMD`.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*hBuffer*	Buffer handle to retrieve information about.
[in]	*iInfoCmd*	Information to be retrieved as defined in `BUFFER_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `BUFFER_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSGetInfo           ( DS_HANDLE        hDataStream,
                                STREAM_INFO_CMD  iInfoCmd,
                                INFO_DATATYPE *  piType,
                                void *           pBuffer,
                                size_t *         piSize )
```

Inquire information about the Data Stream module associated with *hDataStream* as defined in `STREAM_INFO_CMD`.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*iInfoCmd*	Information to be retrieved as defined in `STREAM_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `STREAM_INFO_CMD` and `INFO_DATATYPE`.

[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSQueueBuffer       ( DS_HANDLE       hDataStream,
                                BUFFER_HANDLE   hBuffer )
```

This function queues a particular buffer for acquisition. A buffer can be queued for acquisition any time after the buffer was announced (before or after the acquisition has been started) when it is not currently queued. Furthermore, a buffer which is already waiting to be delivered can not be queued for acquisition. A queued buffer can not be revoked.

The order of the delivered buffers is not necessarily the same as the order in which they have been queued.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*hBuffer*	Buffer handle to queue.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSRevokeBuffer      ( DS_HANDLE       hDataStream,
                                BUFFER_HANDLE   hBuffer,
                                void **         ppBuffer,
                                void **         ppPrivate )
```

Removes an announced buffer from the acquisition engine. This function will free all internally allocated resources associated with this buffer. A buffer can only be revoked if it is not queued in any queue. A buffer is automatically revoked when the stream is closed.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*hBuffer*	Buffer handle to revoke.

[out]	*ppBuffer*	Pointer to the buffer memory. This is for convenience if GenTL Consumer allocated memory is used which is to be freed. If the buffer was allocated by the GenTL Producer NULL is to be returned. If the parameter is set to `NULL` it is ignored
[out]	*ppPrivate*	Pointer to the user data pointer given in the announce function. If the parameter is set to `NULL` it is ignored

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSStartAcquisition  ( DS_HANDLE        hDataStream,
                                ACQ_START_FLAGS  iStartFlags,
                                uint64_t         iNumToAcquire )
```

Starts the acquisition engine on the host.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*iStartFlags*	As defined in `ACQ_START_FLAGS`.
[in]	*iNumToAcquire*	Sets the number of frames after which the acquisition engine stops automatically. If set to 0xFFFFFFFFFFFFFFFF to the acquisition must be stopped manually using the `DSStopAcquisition` function.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  DSStopAcquisition   ( DS_HANDLE        hDataStream,
                                ACQ_STOP_FLAGS   iStopFlags )
```

Stops the acquisition engine on the host.

Parameters

[in]	*hDataStream*	Data Stream module to work on.
[in]	*iStopFlags*	Stops the acquisition as defined in `ACQ_STOP_FLAGS`.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

6.3.6 Port Functions

```
GC_ERROR  GCGetPortInfo      ( PORT_HANDLE     hPort,
                               PORT_INFO_CMD   iInfoCmd,
                               INFO_DATATYPE * piType,
                               void *          pBuffer,
                               size_t *        piSize )
```

Queries detailed port information as defined in `PORT_INFO_CMD`.

Parameters

[in]	*hPort*	Module or remote device port handle to access Port from.
[in]	*iInfoCmd*	Information to be retrieved as defined in `PORT_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `PORT_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  GCGetPortURL       ( PORT_HANDLE     hPort,
                               char *          sURL,
                               size_t *        piSize )
```

Retrieves a URL list with the XML description for the given *hPort*. See 4.1.2 XML Description page 30 for more information about supported URLs. Each URL is terminated with a trailing '\0' and after the last URL are two '\0'.

Parameters

[in]	*hPort*	Module or remote device port handle to access Port from.
[in,out]	*sURL*	Pointer to a user allocated string buffer to receive the list of URLs If this parameter is `NULL`, *piSize* will contain the needed size of *sURL* in bytes. Each entry in the list is 0 terminated. After the last entry there is an additional 0. The size includes the terminating 0 characters.

[in,out]	*piSize*	*sURL* equal `NULL`: out: minimal size of *sURL* in bytes to hold all information *sURL* unequal `NULL`: in: size of the provided *sURL* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  GCReadPort          ( PORT_HANDLE     hPort,
                                uint64_t        iAddress,
                                void *          pBuffer,
                                size_t *        piSize )
```

Reads a number of bytes from a given *iAddress* from the specified *hPort*. This is the global GenICam GenApi read access function for all ports implemented in the GenTL implementation. The endianess of the data content is specified by the `GCGetPortInfo` function.

If the underlying technology has alignment restrictions on the port read the GenTL Provider implementation has to handle this internally. So for example if the underlying technology only allows a uint32_t aligned access and the calling GenTLConsumer wants to read 5 bytes starting at address 2. The implementation has to read 8 bytes starting at address 0 and then it must only return the requested 5 bytes.

Parameters

[in]	*hPort*	Module or remote device port handle to access Port from.
[in]	*iAddress*	Byte address to read from.
[out]	*pBuffer*	Pointer to a user allocated byte buffer to receive data; this must not be `NULL`.
[in,out]	*piSize*	Size of the provided *pBuffer* and thus the amount of bytes to read from the register map; after the read operation this parameter holds the information about the bytes actually read.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  GCWritePort        ( PORT_HANDLE     hPort,
                               uint64_t        iAddress,
                               const void *    pBuffer,
                               size_t *        piSize )
```

Writes a number of bytes at the given *iAddress* to the specified *hPort*. This is the global GenICam GenApi write access function for all ports implemented in the GenTL implementation. The endianess of the data content is specified by the `GCGetPortInfo` function.

If the underlying technology has alignment restrictions on the port write the GenTL Provider implementation has to handle this internally. So for example if the underlying technology only allows a uint32_t aligned access and the calling GenTL Consumer wants to write 5 bytes starting at address 2. The implementation has to read 8 bytes starting at address 0, replace the 5 bytes provided and then write the 8 bytes back (read modify write).

Parameters

[in]	*hPort*	Module or remote device port handle to access Port from.
[in]	*iAddress*	Byte address to write to.
[in]	*pBuffer*	Pointer to a user allocated byte buffer containing the data to write; this must not be `NULL`.
[in,out]	*piSize*	Size of the provided *pBuffer* and thus the amount of bytes to write to the register map; after the write operation this parameter holds the information about the bytes actually written.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

6.3.7 Signaling Functions

```
GC_ERROR  EventFlush         ( EVENT_HANDLE    hEvent )
```

Flushes all events in the given *hEvent* object. This call empties the event data queue.

Parameters

[in]	*hEvent*	Event handle to flush queue on.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  EventGetData        ( EVENT_HANDLE    hEvent,
                                 void *          pBuffer,
                                 size_t *        piSize,
                                 uint64_t        iTimeout )
```

Retrieves the oldest event data from the event data queue associated with the *hEvent*.

The data content can be queried by the `EventGetDataInfo` function.

The default buffer size which can hold all the event data can be queried with the `EventGetInfo` function. This needs to be queried only once. The default size must not change during runtime.

Parameters

[in]	*hEvent*	Event handle to wait for
[out]	*pBuffer*	Pointer to a user allocated buffer to receive the event data. The data type of the buffer is dependent on the event ID of the *hEvent*. If this value is `NULL` the data is removed from the queue without being delivered.
[in,out]	*piSize*	Size of the provided *pBuffer* in bytes; after the write operation this parameter holds the information about the bytes actually written.
[in]	*iTimeout*	Timeout for the wait in ms. A value of 0xFFFFFFFFFFFFFFFF is interpreted as INFINITE. A value of 0 checks the state of the event object and returns immediately either with a timeout or with event data.

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  EventGetDataInfo    ( EVENT_HANDLE    hEvent,
                                 const void *    pInBuffer,
                                 size_t          iInSize,
                                 EVENT_DATA_INFO_CMD iInfoCmd,
                                 INFO_DATATYPE * piType,
                                 void *          pOutBuffer,
                                 size_t *        piOutSize )
```

Parses the transport layer technology dependent event info.

Parameters

[in]	*hEvent*	Event handle to parse data from.
[in]	*pInBuffer*	Pointer to a buffer containing event data. This value must not be `NULL`.

[in]	*iInSize*	Size of the provided *pInBuffer* in bytes
[in]	*iInfoCmd*	Information to be retrieved as defined in `EVENT_DATA_INFO_CMD` and `EVENT_TYPE`.
[out]	*piType*	Data type of the *pOutBuffer* content as defined in the `EVENT_DATA_INFO_CMD`, `EVENT_TYPE` and `INFO_DATATYPE`.
[in,out]	*pOutBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piOutSize* will contain the minimal size of *pOutBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piOutSize*	*pOutBuffer* equal `NULL:` out: minimal size of *pOutBuffer* in bytes to hold all information *pOutBuffer* unequal `NULL:` in: size of the provided *pOutBuffer* in bytes out: number of bytes filled by the function

Returns

`GC_ERROR`: Unequal `GC_ERR_SUCCESS` on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  EventGetInfo       ( EVENT_HANDLE    hEvent,
                               EVENT_INFO_CMD  iInfoCmd,
                               INFO_DATATYPE * piType,
                               void *          pBuffer,
                               size_t *        piSize )
```

Retrieves information about the given *hEvent* object as defined in `EVENT_INFO_CMD`.

Parameters

[in]	*hEvent*	Event handle to retrieve info from.
[in]	*iInfoCmd*	Information to be retrieved as defined in `EVENT_INFO_CMD`.
[out]	*piType*	Data type of the *pBuffer* content as defined in the `EVENT_INFO_CMD` and `INFO_DATATYPE`.
[in,out]	*pBuffer*	Pointer to a user allocated buffer to receive the requested information. If this parameter is `NULL`, *piSize* will contain the minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.
[in,out]	*piSize*	*pBuffer* equal `NULL:` out: minimal size of *pBuffer* in bytes to hold all information *pBuffer* unequal `NULL:` in: size of the provided *pBuffer* in bytes out: number of bytes filled by the function

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  EventKill           ( EVENT_HANDLE    hEvent )
```

Terminate any waiting operation on a previously registered event object.

Parameters

[in]	*hEvent*	Handle to event object.

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

```
GC_ERROR  GCRegisterEvent     ( EVENTSRC_HANDLE hModule,
                                EVENT_TYPE      iEventID,
                                EVENT_HANDLE *  phEvent )
```

Registers an event object to a certain *iEventID*. The implementation might change depending on the platform.

Every event registered must be unregistered with the GCUnregisterEvent function.

Parameters

[in]	*hModule*	Module handle to access to register event to.
[in]	*iEventID*	Event type to register as defined in EVENT_TYPE.
[out]	*phEvent*	New handle to an event object to work with.

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46. If the given *iEventID* has been registered before on the given *hModule* this function returns GC_ERR_RESOURCE_IN_USE.

```
GC_ERROR  GCUnregisterEvent   ( EVENTSRC_HANDLE hModule,
                                EVENT_TYPE      iEventID )
```

Unregisters the given *iEventID* from the given *hModule*.

Parameters

[in]	*hModule*	Module handle to access to register event to.
[in]	*iEventID*	Event type to unregister as defined in EVENT_TYPE.

Returns

GC_ERROR: Unequal GC_ERR_SUCCESS on error. See 6.1.4 Error Handling page 46.

6.4 Enumerations

Enumeration values are signed 32 bit integers.

6.4.1 Library and System Enumerations

`enum INFO_DATATYPE`

Defines the data type possible for the various Info functions. The data type itself may define its size. For buffer or string types the *piSize* parameter must be used to query the actual amount of data being written.

Enumerator	Value	Description
INFO_DATATYPE_UNKNOWN	0	Unknown data type. This value is never returned from a function but can be used to initialize the variable to inquire the type.
INFO_DATATYPE_STRING	1	0-terminated C string (ASCII encoded).
INFO_DATATYPE_STRINGLIST	2	Concatenated INFO_DATATYPE_STRING list. End of list is signaled with an additional 0.
INFO_DATATYPE_INT16	3	Signed 16 bit integer.
INFO_DATATYPE_UINT16	4	Unsigned 16 bit integer.
INFO_DATATYPE_INT32	5	Signed 32 bit integer.
INFO_DATATYPE_UINT32	6	Unsigned 32 bit integer.
INFO_DATATYPE_INT64	7	Signed 64 bit integer.
INFO_DATATYPE_UINT64	8	Unsigned 64 bit integer.
INFO_DATATYPE_FLOAT64	9	Signed 64 bit floating point number.
INFO_DATATYPE_PTR	10	Pointer type (void*). Size is platform dependent (32 bit on 32 bit platforms)
INFO_DATATYPE_BOOL8	11	Boolean value occupying 8 bit. 0 for false and anything for true.
INFO_DATATYPE_SIZET	12	Platform dependent unsigned integer (32 bit on 32 bit platforms)
INFO_DATATYPE_BUFFER	13	Like a INFO_DATATYPE_STRING but with arbitrary data and no 0 termination.
INFO_DATATYPE_CUSTOM_ID	1000	Starting value for Custom IDs

`enum TL_INFO_CMD`

System module information commands for the `TLGetInfo` and `GCGetInfo` functions.

Enumerator	Value	Description

Enumerator	Value	Description
TL_INFO_ID	0	GenTL Producer vendor name. Data type: STRING.
TL_INFO_VENDOR	1	GenTL Producer vendor name. Data type: STRING.
TL_INFO_MODEL	2	GenTL Producer model name. Data type: STRING.
TL_INFO_VERSION	3	GenTL Producer revision. Data type: STRING.
TL_INFO_TLTYPE	4	Transport layer technologies that are supported. • "Mixed" for several technologies • "Custom" for not defined ones • "GEV" for GigE Vision • "CL" for Camera Link • "IIDC" for IIDC 1394 • "UVC" for USB video class devices Data type: STRING.
TL_INFO_NAME	5	File name including extension of the library. Data type: STRING.
TL_INFO_PATHNAME	6	Full path including file name and extension of the library. Data type: STRING.
TL_INFO_DISPLAYNAME	7	User readable name of the GenTL Producer. Data type: STRING.
TL_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

6.4.2 Interface Enumerations

```
enum INTERFACE_INFO_CMD
```

This enumeration defines commands to retrieve information with the `IFGetInfo` function from the Interface module.

Enumerator	Value	Description
INTERFACE_INFO_ID	0	Unique ID of the interface. Data type: STRING
INTERFACE_INFO_DISPLAYNAME	1	User readable name of the interface. Data type: STRING
INTERFACE_INFO_TLTYPE	2	Transport layer technologies that are

Enumerator	Value	Description
		supported. • ”Custom“ for not defined ones • “GEV” for GigE Vision • “CL” for Camera Link • “IIDC” for IIDC 1394 • “UVC” for USB video class devices Data type: STRING.
INTERFACE_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

6.4.3 Device Enumerations

`enum DEVICE_ACCESS_FLAGS`

This enumeration defines flags how a device is to be opened with the `IFOpenDevice` function. Also possible results are defined.

Enumerator	Value	Description
DEVICE_ACCESS_UNKNOWN	0	Not used in a command. Can be used to initialize a variable to query that information.
DEVICE_ACCESS_NONE	1	This either means that the device is not open because it was not opened before or the access to it was denied.
DEVICE_ACCESS_READONLY	2	Open the device read only. All Port functions can only read from the device.
DEVICE_ACCESS_CONTROL	3	Open the device in a way that other hosts/processes can have read only access to the device. Device access level is read/write for this process.
DEVICE_ACCESS_EXCLUSIVE	4	Open the device in a way that only this host/process can have access to the device. Device access level is read/write for this process.
DEVICE_ACCESS_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

`enum DEVICE_ACCESS_STATUS`

This enumeration defines the status codes used in the info functions to retrieve the current accessibility of the device.

Enumerator	Value	Description

Enumerator	Value	Description
DEVICE_ACCESS_STATUS_UNKNOWN	0	The current availability of the device is unknown.
DEVICE_ACCESS_STATUS_READWRITE	1	The device is available for Read/Write access
DEVICE_ACCESS_STATUS_READONLY	2	The device is available for Read access.
DEVICE_ACCESS_STATUS_NOACCESS	3	The device is not available either because it is already open or because it is not reachable.
DEVICE_ACCESS_STATUS_CUSTOM_ID	1000	Starting value for custom IDs.

`enum DEVICE_INFO_CMD`

This enumeration defines commands to retrieve information with the `DevGetInfo` function on a device handle.

Enumerator	Value	Description
DEVICE_INFO_ID	0	Unique ID of the device. Data type: STRING
DEVICE_INFO_VENDOR	1	Device vendor name. Data type: STRING
DEVICE_INFO_MODEL	2	Device model name. Data type: STRING
DEVICE_INFO_TLTYPE	3	Transport layer technologies that are supported. • “Custom“ for not defined ones • “GEV” for GigE Vision • “CL” for Camera Link • “IIDC” for IIDC 1394 • “UVC” for USB video class devices Data type: STRING.
DEVICE_INFO_DISPLAYNAME	4	User readable name of the device. If this is not defined in the device this should be “VENDOR MODEL (ID)” Data type: STRING
DEVICE_INFO_ACCESS_STATUS	5	Gets the access status the GenTL Producer has on the device. Data type: INT32 (`DEVICE_ACCESS_STATUS` enumeration value)
DEVICE_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom Ids.

6.4.4 Data Stream Enumerations

`enum ACQ_QUEUE_TYPE`

This enumeration commands from which to which queue/pool buffers are flushed with the `DSFlushQueue` function.

Enumerator	Value	Description
ACQ_QUEUE_INPUT_TO_OUTPUT	0	Flushes the input pool to the output buffer queue and if necessary adds entries in the "New Buffer" event data queue.
ACQ_QUEUE_OUTPUT_DISCARD	1	Discards all buffers in the output buffer queue and if necessary remove the entries from the event data queue.
ACQ_QUEUE_ALL_TO_INPUT	2	Puts all buffers in the input pool. Even those in the output buffer queue and discard entries in the event data queue.
ACQ_QUEUE_UNQUEUED_TO_INPUT	3	Puts all buffers that are not in the input pool or the output buffer queue in the input pool.
ACQ_QUEUE_ALL_DISCARD	4	Discards all buffers in the input pool and output buffer queue.
ACQ_QUEUE_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

`enum ACQ_START_FLAGS`

This enumeration defines special start flags for the acquisition engine. The function used is `DSStartAcquisition`.

Enumerator	Value	Description
ACQ_START_FLAGS_DEFAULT	0	Default behavior.
ACQ_START_FLAGS_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

`enum ACQ_STOP_FLAGS`

This enumeration defines special stop flags for the acquisition engine. The function used is `DSStopAcquisition`.

Enumerator	Value	Description

Enumerator	Value	Description
ACQ_STOP_FLAGS_DEFAULT	0	Stop the acquisition engine when the currently running tasks like filling a buffer are completed (default behavior).
ACQ_STOP_FLAGS_KILL	1	Stop the acquisition engine immediately and leave buffers currently being filled in the Input Buffer Pool.
ACQ_STOP_FLAGS_CUSTOM_ID	1000	Starting value for GenTL Producer custom Ids.

`enum BUFFER_INFO_CMD`

This enumeration defines commands to retrieve information with the `DSGetBufferInfo` function on a buffer handle.

Enumerator	Value	Description
BUFFER_INFO_BASE	0	Base address of the buffer memory. Data type: PTR
BUFFER_INFO_SIZE	1	Size of the buffer in bytes. Data type: SIZET
BUFFER_INFO_USER_PTR	2	Private data pointer of the GenTL Consumer. Data type: PTR
BUFFER_INFO_TIMESTAMP	3	Timestamp the buffer was acquired. The unit is device/implementation dependent. In case the technology and/or the device does not support this for example under Windows a QueryPerformanceCounter can be used. Data type: UINT64
BUFFER_INFO_NEW_DATA	4	Flag to indicate that the buffer contains new data since the last call. Data type: BOOL8
BUFFER_INFO_IS_QUEUED	5	Flag to indicate if the buffer is in the input pool or output buffer queue. Data type: BOOL8
BUFFER_INFO_IS_ACQUIRING	6	Flag to indicate that the buffer is currently being filled with data. Data type: BOOL8
BUFFER_INFO_IS_INCOMPLETE	7	Flag to indicate that a buffer was filled but an error occurred during that process. Data type: BOOL8
BUFFER_INFO_TLTYPE	8	Transport layer technologies that are supported. • “Custom“ for not defined ones

Enumerator	Value	Description
		• "GEV" for GigE Vision • "CL" for Camera Link • "IIDC" for IIDC 1394 • "UVC" for USB video class devices Data type: STRING
BUFFER_INFO_SIZE_FILLED	9	Number of bytes written into the buffer last time it has been filled. This value is reset to 0 when the buffer is placed into the Input Buffer Pool. Data type: SIZET
BUFFER_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

```
enum STREAM_INFO_CMD
```

This enumeration defines commands to retrieve information with the `DSGetInfo` function on a data stream handle.

Enumerator	Value	Description
STREAM_INFO_ID	0	Unique ID of the data stream. Data type: STRING
STREAM_INFO_NUM_DELIVERED	1	Number of acquired frames since last acquisition start. Data type: UINT64
STREAM_INFO_NUM_UNDERRUN	2	Number of lost frames due to queue underrun. Data type: UINT64
STREAM_INFO_NUM_ANNOUNCED	3	Number of announced buffers. Data type: SIZET
STREAM_INFO_NUM_QUEUED	4	Number of buffers in the input pool. Data type: SIZET
STREAM_INFO_NUM_AWAIT_DELIVERY	5	Number of buffers in the output buffer queue. Data type: SIZET
STREAM_INFO_NUM_STARTED	6	Number of frames started in the acquisition engine. Data type: UINT64
STREAM_INFO_PAYLOAD_SIZE	7	Size of the expected data in bytes. Data type: SIZET
STREAM_INFO_IS_GRABBING	8	Flag indicating whether the acquisition engine is started or not. This is independent from the acquisition status of the remote

Enumerator	Value	Description
		device. Data type: BOOL8
STREAM_INFO_DEFINES_PAYLOADSIZE	9	Flag that indicating that this data stream defines a payload size independent from the remote device. If `false` the size of the expected PayloadSize is to be retrieved from the remote device. If `true` the expected PayloadSize is to be inquired from the Data Stream module. Data type: BOOL8
STREAM_INFO_TLTYPE	10	Transport layer technologies that are supported. • "Custom" for not defined ones • "GEV" for GigE Vision • "CL" for Camera Link • "IIDC" for IIDC 1394 • "UVC" for USB video class devices Data type: STRING
STREAM_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

6.4.5 Port Enumerations

```
enum PORT_INFO_CMD
```

This enumeration defines commands to retrieve information with the `GCGetPortInfo` function on a module or remote device handle.

Enumerator	Value	Description
PORT_INFO_ID	0	Unique ID of the module the port references. Data type: STRING
PORT_INFO_VENDOR	1	Port vendor name. Data type: STRING
PORT_INFO_MODEL	2	Port model name. Data type: STRING
PORT_INFO_TLTYPE	3	Transport layer technologies that are supported. • "Custom" for not defined ones • "GEV" for GigE Vision • "CL" for Camera Link

Enumerator	Value	Description
		• "IIDC" for IIDC 1394 • "UVC" for USB video class devices Data type: STRING.
PORT_INFO_MODULE	4	GenTL Module the port refers to: • "TLSystem" for the System module • "TLInterface" for the Interface module • "TLDevice" for the Device module • "TLDataStream" for the Data Stream module • "TLBuffer" for the Buffer module • "Device" for the remote device Data type: STRING
PORT_INFO_LITTLE_ENDIAN	5	Flag indicating that the port's data is little endian. Data type: BOOL8
PORT_INFO_BIG_ENDIAN	6	Flag indicating that the port's data is big endian. Data type: BOOL8
PORT_INFO_ACCESS_READ	7	Flag indicating that read access is allowed. Data type: BOOL8
PORT_INFO_ACCESS_WRITE	8	Flag indicating that write access is allowed. Data type: BOOL8
PORT_INFO_ACCESS_NA	9	Flag indicating that the port is currently not available. Data type: BOOL8
PORT_INFO_ACCESS_NI	10	Flag indicating that no port is implemented. Data type: BOOL8
PORT_INFO_VERSION	11	Version of the port. Data type: STRING
PORT_INFO_PORTNAME	12	Name of the port as referenced in the XML description. Data type: STRING
PORT_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

6.4.6 Signaling Enumerations

`enum EVENT_DATA_INFO_CMD`

This enumeration defines commands to retrieve information with the `EventGetDataInfo` function on delivered event data.

The availability and the data type of the enumerators depend on the event type (see below).

Enumerator	Value	Description
EVENT_DATA_ID	0	Defines a date in the event data to identify the object or feature the event refers to. This can be e.g. the error code for an error event or the feature name for GenApi related events.
EVENT_DATA_VALUE	1	Defines additional data to an ID. This can be e.g. the error message for an error event.
EVENT_DATA_CUSTOM_ID	1000	Starting value for GenTL Producer custom Ids.

`enum EVENT_INFO_CMD`

This enumeration defines command to retrieve information with the `EventGetInfo` function on an event handle.

Enumerator	Value	Description
EVENT_EVENT_TYPE	0	The event type of the event handle. Data type: INT32 (EVENT_TYPE enum value)
EVENT_NUM_IN_QUEUE	1	Number of events in the event data queue. Data type: SIZET
EVENT_NUM_FIRED	2	Number of events that were fired since the creation of the module. Data type: UINT64
EVENT_INFO_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs.

`enum EVENT_TYPE`

Known event types that can be registered on certain modules with the `GCRegisterEvent` function. See 4.2 Signaling page 32 for more information.

Specific values of the event data can be queried with the `EventGetDataInfo` function. It is stated in the table which enumerators specify values that can be retrieved by a specific event type.

Enumerator	Value	Description
EVENT_ERROR	0	Notification on module errors. Values that can be retrieved are: • EVENT_DATA_ID Data type: INT32 (GC_ERROR) • EVENT_DATA_VALUE Data type: STRING (Description)
EVENT_NEW_BUFFER	1	Notification on newly filled buffers. Values that can be retrieved are: • EVENT_DATA_ID Data type: PTR (Buffer handle) • EVENT_DATA_VALUE Data type: PTR (Private pointer)
EVENT_FEATURE_INVALIDATE	2	Notification if a feature was changed by the GenTL Producer driver and thus needs to be invalidated in the GenICam GenApi instance using the module. Values that can be retrieved are: • EVENT_DATA_ID Data type: STRING (Feature name)
EVENT_FEATURE_CHANGE	3	Notification if the GenTL Producer driver wants to manually set a feature in the GenICam GenApi instance using the module. Values that can be retrieved are: • EVENT_DATA_ID Data type: STRING (Feature name) • EVENT_DATA_VALUE Data type: STRING (Feature value)
EVENT_FEATURE_DEVEVENT	4	Notification if the GenTL Producer wants to inform the GenICam GenApi instance of the remote device that a GenApi compatible event was fired. Values that can be retrieved are: • EVENT_DATA_ID Data type: STRING (Event ID) • EVENT_DATA_VALUE Data type: BUFFER (optional data)
EVENT_CUSTOM_ID	1000	Starting value for GenTL Producer custom events.

6.5 Structures

Structures are byte aligned. The size of pointers as members is platform dependent.

6.5.1 Signaling Structures

```
struct EVENT_NEW_BUFFER_DATA
```

Structure of the data returned from a signaled “New Buffer” event.

Member	Type	Description
BufferHandle	BUFFER_HANDLE	Buffer handle which contains new data.
UserPointer	void *	User pointer provided at announcement of the buffer.

7 Standard Feature Naming Convention for GenTL

The different GenTL modules expose their features through the Port functions interface. To interpret the virtual register map of each module the GenICam GenApi is used. This document only contains the names of mandatory features that must be implemented to guarantee interoperability between the different GenTL Consumers and GenTL Producers. Additional features and descriptions can be found in the GenICam Standard Feature Naming Convention document (SFNC).

For technical reasons the different transport layer technologies and protocols (GigE Vision, IIDC 1394, Camera Link,…) have different feature sets. This is addressed in dedicated sections specialized on these technologies. Also features specific to one technology have a prefix indicating its origin, e.g. Gev for GigE Vision specific features or Cl for Camera Link specific features.

7.1 Common

The common feature set is mandatory for all GenTL Producer implementations and used for all transport layer technologies.

7.1.1 System Module

This is a description of all features which must be accessible in the System module: Port functions use the `TL_HANDLE` to access these features. The Port access for this module is mandatory.

Table 7-1: System module information features

Name	Interface	Access	Description
TLPort	IPort	R/W	The port through which the System module is accessed.
TLVendorName	IString	R	Name of the GenTL Producer vendor.
TLModelName	IString	R	Name of the GenTL Producer to distinguish different kinds of GenTL Producer implementations from one vendor.
TLID	IString	R	Unique identifier of the GenTL Producer like a GUID.
TLVersion	IString	R	Vendor specific version string.
TLPath	IString	R	Full path to the GenTL Producer driver including name and extension.
TLType	IEnumeration	R	Identifies the transport layer technology of the GenTL Producer implementation. Values: • “Mixed” for several technologies • “Custom” for not defined ones

Name	Interface	Access	Description
			• "GEV" for GigE Vision • "CL" for Camera Link • "IIDC" for IIDC 1394 • "UVC" for USB video class devices
GenTLVersionMajor	IInteger	R	Major version number of the GenTL specification the GenTL Producer implementation complies with.
GenTLVersionMinor	IInteger	R	Minor version number of the GenTL specification the GenTL Producer implementation complies with.

Table 7-2: Interface enumeration features

Name	Interface	Access	Description
InterfaceUpdateList	ICommand	(R)/W	Updates the internal interface list. This feature should be readable if the execution can not performed immediately. The command then returns and the status can be polled. This function interacts with the TLUpdateInterfaceList of the GenTL Producer. It is up to the GenTL Consumer to handle access in case both methods are used.
InterfaceSelector	IInteger	R/W	Selector for the different GenTL Producer interfaces. This interface list only changes on execution of InterfaceUpdateList.
InterfaceID [InterfaceSelector]	IString	R	GenTL Producer wide unique identifier of the selected interface. This interface list only changes on execution of InterfaceUpdateList.

7.1.2 Interface Module

All features that must be accessible in the interface module are listed here: Port functions use the `IF_HANDLE` to access these features. The Port access for this module is mandatory.

Table 7-3: Interface information features

Name	Interface	Access	Description
InterfacePort	IPort	R/W	The port through which the interface module is accessed.
InterfaceID	IString	R	GenTL Producer wide unique

Name	Interface	Access	Description
			identifier of the selected interface.
InterfaceType	IEnumeration	R	Identifies the transport layer technology of the interface. Values: • "Custom" for not defined ones • "GEV" for GigE Vision • "CL" for Camera Link • "IIDC" for IIDC 1394 • "UVC" for USB video devices

Table 7-4: Device enumeration features

Name	Interface	Access	Description
DeviceUpdateList	ICommand	(R)/W	Updates the internal device list. This feature should be readable if the execution can not performed immediately. The command then returns and the status can be polled. This function interacts with the TLUpdateDeviceList function of the GenTL Producer. It is up to the GenTL Consumer to handle access in case both methods are used.
DeviceSelector	IInteger	R/W	Selector for the different devices on this interface. This value only changes on execution of DeviceUpdateList.
DeviceID [DeviceSelector]	IString	R	Interface wide unique identifier of the selected device. This value only changes on execution of DeviceUpdateList.
DeviceVendorName [DeviceSelector]	IString	R	Name of the device vendor. This value only changes on execution of DeviceUpdateList.
DeviceModelName [DeviceSelector]	IString	R	Name of the device model. This value only changes on execution of DeviceUpdateList.
DeviceAccessStatus [DeviceSelector]	IEnumeration	R	Gives the current access status. This value only changes on execution of DeviceUpdateList. Values: • "ReadWrite" for full access • "ReadOnly" for read-only access • "NoAccess" if another device has

Name	Interface	Access	Description
			exclusive access

7.1.3 Device Module

Contains all features which must be accessible in the Device module: Port functions use the `DEV_HANDLE` to access these features. The Port access for this module is mandatory.

Do not mistake this Device module Port access with the remote device Port access. This module represents the GenTL Producer's view on the remote device. The remote device port is retrieved via the `DevGetPort` function returning a `PORT_HANDLE` for the remote device.

Table 7-5: Device information features

Name	Interface	Access	Description
DevicePort	IPort	R/W	Port through which the Device module is accessed.
DeviceID	IString	R	Interface wide unique identifier of this device.
DeviceVendorName	IString	R	Name of the device vendor.
DeviceModelName	IString	R	Name of the device model.
DeviceType	IEnumeration	R	Identifies the transport layer technology of the device. Values: • "Custom" for not defined ones • "GEV" for GigE Vision • "CL" for Camera Link • "IIDC" for IIDC 1394 • "UVC" for USB video class devices

Table 7-6: Stream enumeration features

Name	Interface	Access	Description
StreamSelector	IInteger	R/W	Selector for the different stream channels
StreamID [StreamSelector]	IString	R	Device unique ID for the stream, e.g. a GUID.

7.1.4 Data Stream Module

This section lists all features which must be available in the stream module: Port functions use the `DS_HANDLE` to access the features. The Port access for this module is mandatory.

Table 7-7: Data Stream information features

Name	Interface	Access	Description
StreamPort	IPort	R/W	Port through which the Data Stream

Name	Interface	Access	Description
			module is accessed.
StreamID	IString	R	Device unique ID for the data stream, e.g. a GUID.
StreamAnnouncedBufferCount	IInteger	R	Number of announced (known) buffers on this stream.
StreamAcquisitionModeSelector	IEnumeration	R/W	Available acquisition modes of this Stream. Value: “Default” (see chapter 5 Acquisition Engine page 37ff)
StreamAnnounceBufferMinimum [AcquisitionModeSelector]	IInteger	R	Minimal number of buffers to announce to enable selected acquisition mode.
StreamType	IEnumeration	R	Identifies the transport layer technology of the stream. Values: • “Custom” for not defined ones • “GEV” for GigE Vision • “CL” for Camera Link • “IIDC” for IIDC 1394 • “UVC” for USB video class devices

7.1.5 Buffer Module

All features that must be accessible on a buffer if a Port access is provided are listed here. Port functions use the `BUFFER_HANDLE` to access these features. The Port access for the `BUFFER_HANDLE` is not mandatory. Thus all features listed here need not be implemented. If a Port access is implemented on the handle though, all mandatory features must be present.

Table 7-8: Buffer information features

Name	Interface	Access	Description
BufferPort	IPort	R/W	Port through which a specific buffer is accessed.
BufferData	IRegister	R/(W)	Entire buffer data.
BufferUserData	IInteger	R	User data provided by user data pointer at buffer announcement. (see chapter 6.3.5 Data Stream Functions page 59ff)

7.2 GigE Vision

For a GenTL Producer implementation supporting GigE Vision the features defined in this section must also be present. All features described in this chapter have to be added to the modules in the common part and are accessed the same way.

7.2.1 System Module

Table 7-9: GigE Vision system information features

Name	Interface	Access	Description
GevVersionMajor	IInteger	R	Major version number of the GigE Vision specification the GenTL Producer implementation complies to. If the System module has a TLType “Mixed” but supports GigE Vision interfaces this feature must be present.
GevVersionMinor	IInteger	R	Minor version number of the GigE Vision specification the GenTL Producer implementation complies to. If the System module has a TLType “Mixed” but supports GigE Vision interfaces this feature must be present.

Table 7-10: GigE Vision interface enumeration features

Name	Interface	Access	Description
GevInterfaceMACAddress [InterfaceSelector]	IInteger	R	48-bit MAC address of the selected interface.
GevInterfaceDefaultIPAddress [InterfaceSelector]	IInteger	R	IP address of the first subnet of the selected interface.
GevInterfaceDefaultSubnetMask [InterfaceSelector]	IInteger	R	Subnet mask of the first subnet of the selected interface.
GevInterfaceDefaultGateway [InterfaceSelector]	IInteger	R	Default gateway of the selected interface.

7.2.2 Interface Module

Table 7-11: GigE Vision interface information features

Name	Interface	Access	Description

Name	Interface	Access	Description
GevInterfaceGatewaySelector	IInteger	R/W	Selector for the different gateway entries for this interface.
GevInterfaceGateway [GevGatewaySelector]	IInteger	R	IP address of the selected gateway entry of this interface.
GevMACAddress	IInteger	R	48-bit MAC address of this interface.
GevInterfaceSubnetSelector	IInteger	R/W	Selector for the subnet of this interface.
GevInterfaceSubnetIPAddress [GevSubnetSelector]	IInteger	R	IP address of the selected subnet of this interface.
GevInterfaceSubnetMask [GevSubnetSelector]	IInteger	R	Subnet mask of the selected subnet of this interface.

Table 7-12: GigE Vision device enumeration features

Name	Interface	Access	Description
GevDeviceIPAddress [DeviceSelector]	IInteger	R	Current IP address of the GVCP interface of the selected remote device.
GevDeviceSubnetMask [DeviceSelector]	IInteger	R	Current subnet mask of the GVCP interface of the selected remote device.
GevDeviceMACAddress [DeviceSelector]	IInteger	R	48-bit MAC address of the GVCP interface of the selected remote device.

7.2.3 Device Module

Table 7-13: GigE Vision device information features

Name	Interface	Access	Description
GevDeviceIPAddress	IInteger	R	Current IP address of the GVCP interface of the remote device.
GevDeviceSubnetMask	IInteger	R	Current subnet mask of the GVCP interface of the remote device.
GevDeviceMACAddress	IInteger	R	48-bit MAC address of the GVCP interface of the remote device.
GevDeviceGateway	IInteger	R	Current gateway IP address of the GVCP interface of the remote device.